Ejecución segura y con limitación de memoria de programas en Python

Trabajo de Fin de Grado

David Sarnago Ojuel

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2021

Documento maquetado con TEXIS v.1.0.

Este documento está preparado para ser imprimido a doble cara.

Ejecución segura y con limitación de memoria de programas en Python

Safe and memory-limited execution of Python programs

Memoria que presenta para optar al título de Grado en Ingeniería Informática

Dirigido por
Marco Antonio Gómez Martín
Pedro Pablo Gómez Martín

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Junio 2021

Resumen

Los jueces en línea, como por ejemplo *¡Acepta el Reto!*[1](Gómez-Martín y Gómez-Martín, 2017), reciben programas enviados por los usuarios y los ejecutan para comprobar su corrección. Esta ejecución debe ser realizada bajo un entorno seguro, que no ponga en riesgo la máquina del juez, y bajo una restricción de memoria impuesta por cada problema.

Desarrollar un sistema de ejecución seguro con limitación de memoria para programas en *Python* está, como el título indica, formado por dos puntos fundamentales.

La limitación de memoria de un programa permite al entorno restringir el uso total que este puede hacer ejecutándose en su interior. El objetivo no solo es evitar poner en riesgo el mismo entorno debido al alto uso de memoria, sino el de implementar una limitación mucho más restrictiva para los programas que permita, por ejemplo, discriminar soluciones con consumo lineal de memoria.

La ejecución segura permite al entorno ejecutar cualquier tipo de programa sin temer por la integridad de la máquina que lo está ejecutando. Esto se puede conseguir restringiendo las funciones que se pueden emplear dentro de un programa o realizando su ejecución en un entorno que por sí mismo no permita la ejecución de ciertas funciones.

Este trabajo consiste en un estudio e implementación de diferentes formas de abarcar los dos puntos anteriores y unirlos en un solo programa, capaz de ejecutar cualquier tipo de código *Python* de forma segura y con una limitación sobre la memoria máxima que puede utilizar.

Palabras clave: *Python*, ejecución segura, limitación de memoria, *¡Acepta el Reto!*, *Linux*, *chroot*, juez en línea, llamadas al sistema.

[1] `https://aceptaelreto.com/`

Abstract

Online judges like *¡Acepta el Reto!*[1](Gómez-Martín y Gómez-Martín, 2017) receive small programs submitted by the users and run them to check their correctness. This execution must be carried out in a safe environment, which does not put the judge's machine at risk, and under memory restriction imposed by each problem.

Developing a memory-limited secure execution system for *Python* is, as the title says, made up of two essential points.

Limiting the memory of a program allows the environment to restrict the maximum memory that can be used by them inside it. The goal is not only to avoid putting the environment at risk due to high memory usage, but to implement a more restrictive limitation for the programs to discriminate, for example, solutions with linear memory consumption.

Safe execution allows the environment to run any kind of program without worrying about the integrity of the machine that is running it. This can be achieved by restricting the functions that can be used in the programs or by running them inside an environment that doesn't allow the execution of those functions by default.

Este trabajo consiste en un estudio de las diferentes formas de abarcar los dos puntos anteriores y unirlos en un solo programa, capaz de ejecutar cualquier tipo de código *Python* de forma segura y con una limitación sobre la memoria máxima que puede utilizar.

This work consists then on a study and implementation of the different ways of dealing with the last two points and combining them on a single program, capable of executing any kind of *Python* code safely and memory-limited.

Keywords: *Python*, safe execution, memory-limited, *¡Acepta el Reto!*, *Linux*, *chroot*, online judges, system calls.

[1] `https://aceptaelreto.com/`

Índice

Capítulo 1

Introducción

La ejecución segura es un pilar fundamental de los jueces en línea actuales, permitiendo que estos puedan verificar la corrección de un programa sin permitir a los usuarios realizar actividades que puedan comprometer el estado de juez y la máquina que lo ejecuta. Es por ello que todos implementan algún tipo de seguridad al ejecutar los envíos (*sandboxing*).

Por el contrario, la mayor parte de los jueces en línea no implementan una limitación "fuerte" en cuanto a uso de memoria, optando por una limitación global, la misma para todos los problemas independientemente de su tipo. En esta faceta, *¡Acepta el Reto!* es particular, aplicando a cada problema un límite de memoria diferente, adaptado a las particularidades de este y siendo un factor más a tener en cuenta a la hora de resolver el problema.

Esta particularidad del juez ha sido la que ha hecho que *Python* nunca fuese añadido a la lista de lenguajes admitidos en el juez, pues era necesario disponer de algún mecanismo para limitar la memoria usada por los programas en *Python* de forma ajustada para no proporcionar ventajas al usar uno u otro lenguaje.

1.1. Objetivos

El objetivo principal de este proyecto es el de desarrollar un entorno que permita la correcta ejecución de cualquier programa en *Python* de forma segura y con limitación sobre la memoria máxima que puede utilizar para su uso en jueces automáticos.

Para lograrlo, lo hemos desglosado en los siguientes pasos intermedios:

- Realizar un estudio sobre el uso de *Python* como lenguaje en la programación competitiva.
- Estudiar el funcionamiento interno de *Python* en lo relativo al uso de memoria.

- Investigar diferentes formas de medir y limitar la memoria que puede utilizar un programa cualquiera y aplicarlo en programas específicos en *Python*.

- Investigar los diferentes mecanismos y herramientas que nos permitan desarrollar un entorno capaz de ejecutar de forma segura cualquier programa.

- Desarrollar el entorno final combinando los métodos elegidos para la limitación y la seguridad.

- Evaluar el correcto funcionamiento del entorno realizando la ejecución de diferentes programas y observando los resultados que obtenemos.

1.2. Plan de trabajo

El proyecto se iniciará con una con una investigación de todo aquello relacionado con *Python*, la limitación de memoria y la ejecución segura.

Realizaremos un pequeño estudio sobre la viabilidad de *Python* como lenguaje en la programación competitiva e investigaremos diferentes jueces en línea que permitan realizar envíos en este lenguaje, y los resultados que obtuvieron aquellos usuarios que utilizan *Python* como su lenguaje principal.

Investigaremos el uso de memoria de diferentes objetos y estructuras en el lenguaje, creando pequeños programas que nos indiquen el tamaño de estas haciendo uso tanto de la librería estándar de *Python* como de módulos externos. Esta fase se realizará en gran parte con el objetivo de familiarizarnos con los mecanismos internos del propio lenguaje.

Descubriremos formas de medir la memoria máxima que ha usado un programa a lo largo de su ejecución y determinaremos cuál de ellas se utilizará en la implementación. Tendremos que realizar el mismo proceso, esta vez para la limitación de la memoria que un programa puede hacer uso durante su ejecución.

Seguiremos avanzando en el proyecto documentándonos sobre sobre la ejecución segura y obtendremos diferentes mecanismos para su implementación, los cuáles a su vez seguramente requieran una nueva fase de investigación.

Una vez tengamos resueltos los dos puntos anteriores, los uniremos en un solo programa, que se encargue de ejecutar los programas en *Python* que le especifiquemos con las limitaciones tanto en tiempo como en memoria que le indiquemos y todo ello en un entorno seguro.

Finalmente, con el sistema de ejecución completamente implementado y en funcionamiento, realizaremos una fase de validación para comprobar que cumple con los objetivos que nos hemos propuesto.

1.3. Estructura de la memoria

Esta memoria está estructurada en 7 capítulos:

1. En este primero planteamos el problema para el cuál queremos desarrollar una solución, así como el plan que se diseñó para conseguirlo.

2. En el segundo capítulo plasmamos la gran mayoría de los conocimientos adquiridos durante la fase de investigación y estudio. Además se introducen algunos de los mecanismos y herramientas que serán utilizados en los siguientes capítulos para el desarrollo del entorno final.

3. En el tercer capítulo se muestran los métodos utilizados para asegurar el sistema de ejecución, con una introducción dónde son explicados, un análisis de lo necesario para aplicarlo concretamente a *Python* y una implementación.

4. El cuarto capítulo profundiza sobre algunos aspectos explorados en el segundo respecto a la limitación de memoria, así como la implementación final para el entorno. Además, se realiza un estudio de la viabilidad del lenguaje para la resolución de problemas limitados en memoria.

5. El quinto capítulo muestra la implementación final, que combina todo lo desarrollado en los dos capítulos anteriores. Ilustra las características del sistema de ejecución, así como pruebas realizadas sobre este.

6. En el sexto capítulo nos dedicamos a probar las capacidades del entorno desarrollado, realizando una fase de validación.

7. Y finalmente, en el séptimo capítulo terminamos el documento con unas conclusiones generales sobre el entorno, sus carencias y sus fortalezas.

Introduction

Safe execution is a cornerstone of current online judges, allowing them to verify the correctness of a program while restricting the users to carry out anything that may compromise the status of the judge and the machine that runs it. Thats why all of them implement some kind of security while executing the submissions (*sandboxing*).

In contrast, the majority of online judges do not implement a "strong" limitation of memory use, choosing instead for a global limit, the same for all problems regardless of their type. In this way, *¡Acepta el Reto!* is one of a kind, enforcing each problem a unique memory-limit, adapted to the kind of problem and its needs, and the limitation itself being another factor to consider while solving the problem.

¡Acepta el Reto! uniqueness of strongly limiting the memory of each problem is the reason that *Python* was never added to the list of available languages on the judge, as it is necessary to have some system to limit the memory used by *Python* programs in order to not provide advantages to use one or another language.

Goals

The main goal of this project is the development of an environment that allows the correct execution of any *Python* program, safely and memory-limited.

In order to achieve it, we need to:

- Conduct a study on the use of *Python* as a language in competitive programming.
- Study the inner workings of *Python* regarding to its memory-use.
- Investigate different ways of measuring and limiting the memory that can be used by any program and apply them to *Python*.
- Investigate different utilities and tools that can be used to develop an environment able to safely run any program.

- Develop the final environment combining the chosen methods for memory-limiting and security.

- Validate the environment by running a set of problems on it and evaluating the results obtained.

Work plan

This project will begin with a general investigation of the language itself, *Python*, memory limitation and safe execution.

We will conduct a small study on the viability of *Python* as a language in competitive programming. Different online judges that allow the use of *Python* will be investigated, obtaining the results of those users who used *Python* as their main language.

We will investigate the memory usage of different objects and structures of the language, coding small programs that show us their sizes, using both *Python* standard library and external modules. This phase will be conducted in order to familiarize ourselves with the internal mechanisms of the language.

The next step will probably be to measure the maximum memory used by a program throughout its execution and to decide which tools will be used for the implementation. Then we repeat this process, this time to figure out how to limit the maximum memory usable by a program during its execution.

Then, documentation about safe execution will begin, obtaining a some mechanisms for the implementation, which will certainly need another investigation phase.

Once we have the last two points in order, we will have to combine both into a single one in charge of the execution of *Python* programs as told with limitations in both time and memory, all of this in a safe environment.

Finally, with the execution system fully developed and working, a validation phase will be conducted in order to evaluate that it meets the objetives.

Memory structure

This memory is structured in 7 chapters:

- In this first one, we present the problem for which we want to develop a solution, as well as this plan followed to achieve it.

- In the second chapter we reflect the majority of the knowledge acquired during the research phase. In addition, some of the mechanisms and tools that will be used in the next chapters are introduced.

- In the third one, we show the methods used to secure the execution system, with an introduction were they are explained, an analysis of its needs to be applied specifically to *Python* and an implementation.
- The fourth chapter goes into great depth on some aspects explored in the second regarding memory limitation, as well as the final implementation for the environment. Also, a study of the viability of the language is carried out for the solving of memory-limited problems.
- The fifth chapter shows the final implementation, which combines everything developed in the previous two chapters. It presents the characteristics of the execution system and some tests performed on it.
- In the sixth one we test the capabilities of the environment performing a validation phase.
- And finally, in the seventh chapter we end this document with some general conclusions about the final environment, its shortcomings and its strengths.

Capítulo 2

Estado del arte

Python es un lenguaje de programación sorprendentemente antiguo (la versión 1.0 fue lanzada en 1991, 5 años antes que Java 1.0) cuya popularidad ha sufrido una explosión en los últimos años debido a su simpleza, facilidad de uso, versatilidad y potencia. Esta popularidad ha sido propulsada en gran parte por su gran variedad de librerías (*Numpy*, *Tensorflow*, *Scikit*, *Pandas*, etc) que lo han hecho el lenguaje por defecto del análisis de datos.

2.1. Programación competitiva

La programación competitiva es un deporte en el que los participantes intentan resolver el mayor número de problemas en el menor tiempo posible. La resolución de los problemas requiere la codificación de un programa acorde a sus características.

Por lo general, los problemas son de naturaleza matemática o lógica y requieren conocimientos de diversos campos como: combinatoria, teoría de números, teoría de grafos, estructuras de datos, algoritmia, etc.

Independientemente del tipo de problema, el proceso de resolución se puede dividir en dos pasos: la construcción de un algoritmo *eficiente* que resuelva el problema, y la implementación de este en un lenguaje de programación. La evaluación de la corrección de los programas se realiza de forma automática en los llamados jueces en línea, que veremos en la siguiente sección.

Cada problema, generalmente, está formado por varias partes:

- *Nombre y límites*: el título del problema, que puede o no dar pistas sobre su resolución, y los límites que se establecen en la ejecución de este, tiempo y memoria.

- *Descripción del problema*: donde se nos plantea el problema que tenemos que resolver. Las descripciones de los problemas varían, desde

aquellas que nos indican directamente la resolución del problema, hasta las que lo ocultan lo máximo posible.

- *Descripción de la entrada y la salida*: la primera nos indica el formato que sigue la entrada, así como los límites que debemos esperar para los parámetros del problema. Esto último es especialmente relevante en la construcción del algoritmo, porque la magnitud de cada parámetro nos da pistas sobre cómo debemos resolverlo (si el problema requiere ordenar y nos dicen que esperemos hasta 1 millón de elementos, no podemos resolverlo con un algoritmo cuadrático, ya que vamos a obtener TLE). La descripción de la salida nos dice cómo espera el juez que esta esté formateada. Dado que la comprobación se hace automáticamente, debemos ser cuidadosos de seguirla exactamente, u obtendremos un veredicto incorrecto.
- *Ejemplos de entrada y salida*: en estas secciones se muestra una entrada y salida de ejemplo, siguiente las descripciones dadas. Estas sirven para comprobar la corrección del programa una vez ha sido codificado.

2.2. Jueces en línea

Los jueces en línea son sistemas en línea que generalmente contienen un repositorio de problemas a resolver y un sistema de evaluación de programas. Sobre ellos, los usuarios pueden enviar sus soluciones a los diferentes problemas y obtener un veredicto sobre su corrección. El sistema compila el código enviado (en caso de tratarse de un lenguaje compilado) y lo ejecuta con una serie de casos secretos a los cuales el usuario no tiene acceso. Las salidas del programa enviado son comparadas con las correctas y en base a ello el juez dicta el veredicto.

No solo se comprueba la corrección de los programas, sino también que estos consiguen la respuesta en un tiempo de ejecución y uso de memoria máxima limitado.

Los posibles veredictos varían con cada juez, aunque los más habituales son:

- *Aceptado* (AC): la salida del programa coincide con la esperada.
- *Respuesta incorrecta* (WA): la salida no coincide con la esperada.
- *Límite de tiempo excedido* (TLE): el programa ha tardado demasiado tiempo en finalizar y se ha terminado su ejecución.
- *Límite de memoria excedido* (MLE): el programa ha utilizado más memoria de la permitida y se ha terminado su ejecución.
- *Error de ejecución* (RTE): el programa ha fallado durante la ejecución.

- *Error de compilación* (CE): el programa enviado no ha podido ser compilado.

En la actualidad existen multitud de jueces en línea, como el ya mencionado *¡Acepta el Reto!*, *Codeforces*[1], *onlinejudge*[2] y muchos otros que comentaremos más adelante.

2.3. Python como lenguaje de programación competitiva

La popularidad de *Python* en otros terrenos no se ha visto reflejada en la programación competitiva y en los jueces automáticos, donde *Python* es un lenguaje de nicho, muchas veces incapaz de resolver problemas por falta de velocidad.

Codeforces es uno de los jueces automáticos más populares de la actualidad. No solo es un juez, sino que también organiza multitud de concursos de programación online. Los resultados de los usuarios en estos concursos (separados en divisiones en base a su dificultad) determinan su *rating*, en un sistema similar al Elo[3](Elo, 1978). Dependiendo del *rating*, los usuarios son clasificados en diferentes categorías(Looi, 2018).

Concretamente, aquellos con un *rating* mayor a 2400 (la categoría *Grandmaster* y superiores), forman el 0.3 % de los usuarios de la página y son conocidos como *Reds*. Un análisis realizado por uno de los usuarios en 2017[4] refleja cómo de los 278 *Reds* que había en ese momento en el juez, solo 3 de ellos han llegado a utilizar *Python* en algún momento. Además recalca que ninguno lo utiliza como su lenguaje principal.

Vemos algo similar en el juez de la Universidad de Valladolid, *UVa Judge* (actualmente *onlinejudge*)(Revilla et al., 2008). El soporte para envíos en *Python* fue añadido en 2016, año en el que tuvo 27.633 envíos (1.45 % del año). Desde entonces, ese porcentaje ha ido aumentando cada año, hasta el 5.49 % en lo que llevamos de 2021[5].

A pesar de este aumento del uso de *Python*, si analizamos estos envíos y los agrupamos por problemas, podemos comprobar como de los casi 5000 problemas disponibles en el juez, alrededor de 2700 de ellos no tienen ningún envío en *Python* correcto (Veredicto AC). Algunos de estos problemas, como el 929 - "Number Maze"[6] con 248 envíos con veredicto límite de tiempo (TLE), probablemente sean imposibles de resolver en el tiempo establecido.

[1] `https://codeforces.com/`

[2] `https://onlinejudge.org/`

[3] `https://en.wikipedia.org/wiki/Elo_rating_system`

[4] `https://codeforces.com/blog/entry/55177`

[5] `https://web.archive.org/web/20210303221028/https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=23`

[6] `https://onlinejudge.org/external/9/929.pdf`

2.4. Jueces automáticos que soportan Python

Como hemos comentado, muchos jueces automáticos soportan el uso de *Python*, entre los ya mencionados tenemos a *UVa judge* y *Codeforces*, además de otros como *Kattis*[1], *SPOJ*[2], *Codechef*[3], *Hackerrank*[4], *Hackerearth*[5], *URI Online Judge*[6], etc.

En todos ellos, el lenguaje sufre las mismas desventajas comentadas en la sección anterior, donde su falta de velocidad hace su uso subóptimo frente a otros lenguajes. Algunos de estos jueces intentan contrarrestar esta desventaja estableciendo un límite de tiempo diferente para cada lenguaje. Por ejemplo, *URI Online Judge* establece un tiempo límite base para C/C++ y a este se le añade tiempo según el lenguaje (a *Python* se le añade 1 segundo). También tenemos a *Codechef* y *Hackerearth*, donde en vez de sumar, los diferentes lenguajes tienen un multiplicador de tiempo sobre el base.

Pasando del tiempo a la memoria, ninguno de estos jueces hace una limitación sobre el uso de memoria tan restrictivo como lo hace *¡Acepta el Reto!* en los lenguajes que soporta, optando la mayoría por una limitación global sobre todos los problemas que debería ser suficiente para cualquier tipo de problema. En *Kattis* por ejemplo, la mayoría de problemas tienen 1024 MB, por lo que el veredicto *MLE* está más como protección al juez que como limitación de los problemas. Si lo comparamos con *¡Acepta el Reto!*, tenemos unos límites de memoria mucho más variados y ajustados, como podemos observar en la siguiente tabla:

Límite (MiB)	2	4	8	10	12	16	20	24	32	48	64	96	128
Nº Problemas	7	399	49	17	1	22	4	1	8	1	4	1	1

Vemos que la gran mayoría tienen un límite de 4 MiB, con los siguientes valores por número de problemas de 8, 16 y 10 MiB. A pesar de estos límites mucho más restrictivos, gran parte de ellos no están limitados para aumentar la dificultad del problema, sino que en su mayoría se les asigna una cantidad adecuada a su resolución. Si un problema requiere leer un número y escribir su cuadrado no vamos a permitir que use 128 MiB.

En contraste, tenemos problemas donde casi el 30 % de los envíos resultan en *MLE* (189 - "Embarque en un transatlántico" y 248 - "Los premios de las tragaperras") con límites de memoria de 8 MiB y 2 MiB respectivamente. En este caso la restricción de memoria sí que hace su resolución de este tipo de problemas más compleja, añadiendo profundidad y variedad a los tipos de problemas ofrecidos.

[1] `https://open.kattis.com`
[2] `https://www.spoj.com/`
[3] `https://www.codechef.com/`
[4] `https://www.hackerrank.com/`
[5] `https://www.hackerearth.com/`
[6] `https://www.urionlinejudge.com.br/`

Cambiando de jueces en línea a *software* para gestión de concursos, *DOMjudge*[1] es un juez diseñado específicamente para el desarrollo de concursos de programación competitiva. Tiene soporte para *Python* y es código abierto, *hosteado* en *github*[2]. Gracias a esto, podemos ver exactamente cómo maneja la ejecución segura de los programas en Python y ver de qué forma limita su uso de memoria. Siendo más específicos, la limitación que impone este juez es la misma que veremos en la sección 2.8.

2.5. Implementación de *Python*

Por lo general, siempre que se habla de *Python* se hace en referencia a la implementación estándar del mismo, *CPython*(van Rossum, 2010). Esta es la implementación de referencia del lenguaje y por mucho la más utilizada. A lo largo de este proyecto, todas las peculiaridades que hemos visto y que veremos son acerca de esta implementación.

La implementación *CPython* consiste en un intérprete del lenguaje escrito en *C*. Antes de interpretarlo, el código *Python* es compilado a *bytecode*(van Rossum, 2010), el cuál sí es interpretado.

Sin embargo, *CPython* no es la única implementación de *Python* existente. Tenemos una implementación escrita en *Java* para su máquina virtual, Jython[3](Juneau et al., 2010); una implementación en *C#*, *IronPython*[4](Foord y Muirhead, 2009); y una versión escrita en el propio *Python*, *PyPy*[5](Bolz et al., 2009). Es esta última en la que nos vamos a centrar en este apartado.

En contraste a la interpretación del código en *CPython*, *PyPy* utiliza una compilación en tiempo de ejecución, *JIT* o *just-in-time*, en donde el código obtenido en la compilación, *bytecode*, es traducido a código máquina durante la propia ejecución. Este cambio hace que, por lo general, *PyPy* sea considerablemente más rápido que *CPython*(Roghult, 2016). Esta velocidad es proporcionada a cambio de una menor compatibilidad con algunas librerías muy usadas del lenguaje, lo que hace que esta implementación alternativa no tenga un uso más frecuente[6].

Esto por el contrario no afecta en absoluto a la programación competitiva, pues esas librerías no se usan en los concursos. Es por esto en gran parte de los jueces en línea *PyPy* sea la implementación utilizada para ejecutar código en *Python*. Por ejemplo, el juez *Kattis* utiliza *PyPy3*, *Codeforces* ofrece *PyPy2* y *PyPy3*, al igual que *Hackerrank*. En este último, ambos *PyPy2* y *PyPy3* tienen un tiempo límite de 4 segundos, frente a las 10 impuestos en las

[1] `https://www.domjudge.org/`
[2] `https://github.com/DOMjudge/domjudge`
[3] `https://www.jython.org/`
[4] `https://ironpython.net/`
[5] `https://www.pypy.org/index.html`
[6] `https://doc.pypy.org/en/latest/cpython_differences.html#extension-modules`

versiones de *CPython*, lo que muestra la superior velocidad esperada de esta implementación.

Por estas razones se ha optado por, al menos, que el sistema de ejecución que se desarrolle en este proyecto cuente con la opción de elegir libremente la implementación de *Python* que ejecutará los programas indicados.

2.6. Tamaño de los datos en memoria

El uso de memoria de *Python* es una de las partes del lenguaje que menos atención ha tenido. A pesar de esta falta de estudio, tenemos a nuestra disposición diferentes herramientas para la medición de la memoria usada por un proceso *Python*.

2.6.1. Librería estándar de Python

Antes de medir la memoria total usada por el proceso, vamos a explorar las opciones que nos ofrece la librería estándar en respecto al uso de memoria.

En primer lugar hemos investigado la función `getsizeof` del módulo `sys`. Esta función acepta como parámetro cualquier objeto *Python* y nos devuelve su tamaño en bytes. En las siguientes tabla se muestran los tamaños de los tipos de datos primitivos y estructuras estándar vacías:

Objeto	int(0)	int(1)	int(10^100)	float(0)	float(1)	float(10^100)
Tamaño (bytes)	24	28	72	24	24	24

Lo primero que nos encontramos es que el tamaño de lo que conocemos en otros lenguajes como *int* o enteros, empieza en 24 bytes. Esto es debido a que en *Python* todos los números están implementados como una clase de números de precisión infinita, con un puntero de 8 bytes a la clase, un contador de referencia también de 8 bytes, y el resto la representación del número. En esta, cada número esta formado por un *array* de enteros de 32 bits, cada uno representando 30 dígitos binarios del número completo. No obstante, en algunas implementaciones (Van Rossum et al., 2000), se utilizan *unsigned short* de 16 bits que representan 15 dígitos binarios cada uno.

El cero en particular está representado por un *array* vacío, lo que hace que ocupe *4 bytes* (32 bits) menos que el 1. De esta forma, los números entre el 1 y el $2^{30} - 1$ ocuparán 28 bytes, los que estén entre el 2^{30} y $2^{60} - 1$ ocuparán 32, etc.

Al final de la tabla tenemos los números reales, en *Python*, *float*. Vemos de nuevo como el 0 vuelve a ocupar 24 bytes y que, al contrario que en los enteros, todos ocupan lo mismo. Esto es debido a que la clase *float* está implementada usando números en punto flotante de doble precisión, de 8 bytes y que al contrario de los enteros, estos no tienen precisión infinita.

Objeto	bool(False)	bool(True)	str(“”)	str(“a”)	str(“abcde”)
Tamaño (bytes)	28	24	49	50	54

Lo único destacable en los *bool* es que el espacio que ocupan *True* y *False* difiere, aunque podemos entender esta diferencia rápidamente si sabemos que *bool* es una subclase de *int*, con *False* siendo 0 y *True* siendo cualquier otro valor.

En el último tipo primitivo, tenemos las cadenas de caracteres, el tipo *string*. En este caso, la cadena vacía ocupa un total de 49 bytes, mientras que cada carácter extra añade 1 byte a este valor.

Merece la pena mencionar que *Python* utiliza la representación Unicode[1](Van Rossum et al., 2000). Como este contiene actualmente 143,859 caracteres, es necesario un mínimo de 18 dígitos binarios para representar cada uno de ellos, que por términos de rendimiento y alineamiento se suelen codificar con 4 bytes. Para reducir el consumo de memoria, *Python* utiliza 3 representaciones de cadenas diferentes según los caracteres que estén contenidos en el *string*:

- 1 byte por caracter: *Latin-1 encoding*, una codificación formada por ASCII y 128 caracteres extra de diferentes lenguajes (como por ejemplo la $\tilde{N}$). Es el caso descrito en la tabla anterior.
- 2 bytes por caracter: *UCS-2 encoding*, contiene casi la totalidad de los caracteres usados en el resto de lenguajes. En este caso, la cadena vacía ocupa 74 bytes.
- 4 bytes por caracter: *UCS-4 encoding*, que contiene la totalidad de Unicode. Utilizado por ejemplo para *emojis*. La cadena vacía en esta representación ocupa un total de 76 bytes.

Objeto	list()	list([1])	list([1, 2])	dict()	dict({1: 1})
Tamaño (bytes)	56	64	72	232	232

Vemos que la lista vacía ocupa 56 bytes. A continuación tenemos la lista que contiene 1 elemento, con un tamaño de 64 bytes. Si recordamos de la primera tabla, el entero *1* ocupa un total de 28 bytes. De la misma forma, tenemos que recordar que los tipos en *Python* son instancias de clases, por lo que llegamos a la conclusión de que la lista no guarda los elementos en sí, sino punteros a estas clases (en *Python* todos los punteros ocupan 8 bytes).

Algo similar ocurre en el caso de los diccionarios. El vacío ocupa 232 bytes, la misma cantidad que un diccionario con un elemento. Esto es debido a que la función `getsizeof` únicamente tiene en cuenta la memoria usada

[1] `https://home.unicode.org/`

directamente por el objeto, por lo que en el caso de las lista solo cuenta el tamaño de la lista en sí más los punteros y en los diccionarios solo cuenta el tamaño de la tabla interna y no el de las claves y valores.

Para obtener el tamaño real de los objetos utilizaremos librerías externas que comentaremos en la siguiente sección.

2.6.2. Módulos externos

El primer módulo que vamos a comentar es ***Remember Me***[1], que surge como alternativa para el método `getsizeof` visto en la sección anterior. Al contrario que este último, *Remember Me* no solo contabiliza la memoria usada por el objeto que queremos medir, sino también de la ocupada por sus los objetos a los que referencia. Si realizamos de nuevo las medidas de la última tabla utilizando este módulo obtenemos:

Objeto	list()	list([1])	list([1, 2])	dict()	dict({1: 1})
Tamaño (bytes)	56	92	128	232	260

Se puede observar que los tamaños son consistentes con lo visto anteriormente, la lista de 1 elemento ocupa 56 bytes de la lista vacía, 8 bytes del puntero y 28 bytes que ocupa el entero.

Una vez que tenemos en nuestras manos una forma de medir el tamaño exacto de los objetos y estructuras, nos encontramos con otra de las peculiaridades de *Python*. Si hacemos una lista de varios enteros iguales (por ejemplo, `[1, 1, 1, 1]`) y obtenemos su tamaño empleando *Remember Me*, se nos devuelve 116 bytes, que son resultado de 56 bytes de la lista vacía, 4 punteros a los elementos de la lista y 28 bytes restantes, que corresponden al tamaño de un entero. Esto se debe a que *Python* reutiliza el mismo objeto (en este caso un entero) en todas las posiciones de la lista, por lo que solo debe instanciar un objeto.

Para comprobar que esta reutilización es cierta y no que la librería es incorrecta, podemos consultar directamente el *id* del objeto, con la función `id()` del mismo *Python*. Haciendo esto, comprobamos que el *id* de todos los elementos de la lista son el mismo.

Además de esta reutilización, *Python* va un paso más allá y establece internamente una caché de enteros pequeños[2](Lutz, 2013) para cada valor entre -5 y 256. Esto significa que, por ejemplo, todos los ceros de un programa *Python* son el mismo objeto al que todas estas referencias apuntan. Este mecanismo de ahorro no es único de *Python*, pues *Java* realiza también el mismo cacheo de enteros pequeños, en el rango de -128 a 127[3].

[1] `https://github.com/liwt31/remember-me`

[2] `https://docs.python.org/3/c-api/long.html#c.PyLong_FromLong`

[3] `https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.1.7`

Comprobar esto último es más complicado de lo que pueda parecer por lo comentado en el párrafo anterior, dado que *Python* también cachea objetos si ve que se están reutilizando. Una posible forma de hacerlo es engañar al intérprete con una función de este estilo:

```
import random
def getX(x):
    return random.randrange(x, x+1)
lista_1 = [1 for a in range(4)]
lista_1_getX = [getX(1) for a in range(4)]
lista_257 = [257 for a in range(4)]
lista_257_getX = [getX(257) for a in range(4)]
```

En el código, definimos la función `getX(x)`, la cuál recibe un entero y devuelve un número aleatorio entre `x` y `x+1` obtenido por la función `randrange`. El engaño consiste en que esta función es cerrada por la izquierda y abierta por la derecha, por lo que siempre va a devolver `x` y el intérprete no va a ser capaz de anticiparse a ello, como comprobamos de forma empírica.

A continuación creamos 4 listas de 4 enteros cada una. Las dos primeras con un entero en el rango de la caché (el 1 por ejemplo) y las dos siguientes con uno fuera (como el 257). La primera de cada par de listas es generada con el número tal cual, esperando que el intérprete cachee los 257; y en las segundas llamando a la función `getX()`, con el objetivo de que no sea capaz de hacerlo.

Si medimos el tamaño de cada lista con *Remember Me* obtenemos:

- `lista_1`: 116 bytes, correspondientes a 56 de la lista vacía, 32 a los 4 punteros y 28 a la instancia de la clase entera con el uno.
- `lista_1_getX`: 116 bytes, exactamente igual que la lista estándar, ya que el cacheo de enteros pequeños entra en acción.
- `lista_257`: 116 bytes, exactamente igual que las listas anteriores, ya que *Python* ha cacheado el 257 de forma independiente.
- `lista_257_getX`: 200 bytes, hemos conseguido "engañar" al intérprete. El tamaño corresponde a 56 bytes de la lista vacía, 32 a los 4 punteros y 112 restantes a las 4 instancias de la clase con el 257.

2.7. Medición del consumo

Una vez analizado el espacio en memoria ocupado por elementos individuales, pasamos a analizar las formas de medir la memoria total usada por el programa *Python* completo.

El primer módulo que vamos a estudiar con esta funcionalidad es ***psutil***[1].

[1] `https://psutil.readthedocs.io/en/latest/`

Este nos permite consultar diferentes métricas de nuestro sistema, como el consumo actual de CPU, uso de discos, de red y muchas otras. Entre ellas, nos interesa especialmente los relacionados con el uso de memoria, que obtendremos usando la función `memory_full_info()` de la librería.

Definimos la siguiente función para consultar, de forma sencilla, el uso de memoria actual del proceso:

```
def memory_usage_psutil():
    process = psutil.Process(os.getpid())
    mem = process.memory_full_info()[0] / float(2 ** 20)
    print(f"Current memory: {mem:.2f} (MB)")
```

y podemos probar su funcionamiento con el siguiente código, en el que lo enfrentamos a la peculiaridad anteriormente vista de la caché de números pequeños (de -5 a 256):

```
memory_usage_psutil()
l = [getX(-10) for a in range(1000000)]
memory_usage_psutil()
l = [getX(10) for a in range(1000000)]
memory_usage_psutil()
```

y su salida:

```
$ python memory_psutil.py
Current memory: 12.03 (MB)
Current memory: 50.99 (MB)
Current memory: 20.46 (MB)
```

Con esto obtenemos, en primer lugar la memoria que usa *Python* para iniciarse, que es de alrededor de 12 MB (de estos 12, alrededor de 8 son utilizados por el intérprete para ejecutar el código, y los 4 restantes son la sobrecarga que introduce el módulo *psutil*), el de una lista de 1 millón de enteros fuera del rango de los cacheados, que es de alrededor de 51 MB y el de una lista de la misma longitud pero con enteros en el rango de la caché interna, que es de alrededor de 20 MB.

Esto concuerda con la información que ya conocemos, la primera lista ocupa un total de 51 MB, de los cuales 12 pertenecen a la carga del programa, 8 MB corresponderían a los punteros de la lista y 28 MB corresponderían a las instancias de todos los elementos, lo cual suma unos 48 MB, cerca de lo obtenido. En la segunda lista, solo tenemos en cuenta la ocupación del intérprete y los punteros de la lista, que suma 20 MB, ya que *10* está en el rango de los enteros cacheados por *Python*, por lo que todos los punteros de la lista apuntan a la misma instancia y no ocupan memoria extra.

El siguiento módulo que vamos a investigar es conocido como **Guppy3**[1].

[1] `https://zhuyifei1999.github.io/guppy3/`

Este módulo nace como un fork de *Guppy-PE*[2] con el objetivo de portarlo a *Python3*. *Guppy3* está formado por dos paquetes, *Heapy*(Nilsson, 2006) y *GSL*. Entre ambos, solo nos interesa el primero. *Heapy* nos permite explorar el estado del *heap* de un programa *Python* en cualquier momento de la ejecución, mostrándonos un resumen del estado actual.

Para visualizar su funcionamiento, vamos a reutilizar una vez más las listas anteriores, esta vez haciendo llamadas a la función `memory_heapy()`, que nos mostrará la ocupación del *heap* en cada momento:

```
def memory_heapy():
    print(h.heap())

memory_heapy()

l = [getX(-10) for a in range(1000000)]
memory_heapy()

l = [getX(10) for a in range(1000000)]
memory_heapy()

l = []
memory_heapy()
```

Y su salida (recortada, ya que el *heap* de *Python* tiene más objetos además de estas listas):

```
$ python memory_heapy.py
Partition of a set of 38545 objects. Total size = 4537771 bytes.
    Index Count   %    Size   % Cumulative % Kind
        0  11170 29 993059 22 993059 22 str
        [... otras entradas no mostradas]

Partition of a set of 1038546 objects. Total size = 40986991 bytes.
    Index Count   %    Size   % Cumulative % Kind
        0 1001218 96 28037464 68 28037464 68 int
        1    104  0  8468120 21 36505584 89 list
        [... otras entradas no mostradas]

Partition of a set of 38546 objects. Total size = 12987239 bytes.
    Index Count   %    Size   % Cumulative % Kind
        0    104  0  8468120 65 8468120 65 list
        1  11170 29   993059  8   9461179 73 str
        [... otras entradas no mostradas]

Partition of a set of 38546 objects. Total size = 4538567 bytes.
```

[2] http://guppy-pe.sourceforge.net/

```
 Index  Count   %     Size   % Cumulative  % Kind
     0  11170  29   993059  22    993059 22 str
     [... otras entradas no mostradas]
```

En primer lugar, vamos a analizar los datos.

- La primera salida corresponde al análisis del *heap* que utiliza el intérprete de *Python* por defecto. Un total de 38.545 objetos con tamaño acumulado de unos 4 MiB. El 29 % de estos objetos son cadenas (*str*), que suman el 22 % del tamaño total.
- En la segunda, que añade una lista de 1 millón de `-10`, vemos como el total de objetos a pasado a 1.038.546, de los cuales 38.545 son parte del intérprete como hemos visto, 1.000.000 son los elementos de la lista y el elemento restante es la lista en sí. Esta vez podemos ver que los *int*, elementos de la lista, constituyen el 96 % de los objetos totales y el 68 % del tamaño, mientras que la lista en sí ocupa el 21 % del tamaño (recordemos que los punteros de la lista cuentan en el tamaño de esta, por eso esta ocupa 8 MiB).
- En la tercera tenemos el resultado del análisis de la lista de 1 millón de `10`. La lista ocupa de nuevo la misma cantidad que la anterior (ya que ambas son idénticas, solo cambian los elementos a los que apuntan sus elementos), pero los *int* no aparecen por ninguna parte, como debe ser debido a la caché de números pequeños de *Python*.
- En la última lectura comprobamos la acción del recolector de basura, reasignando la lista `l` a una lista vacía y midiendo la memoria utilizada actualmente. Vemos cómo este actúa de forma instantánea, limpiando todo el contenido de la lista y bajando el consumo de memoria total de nuevo a unos 4 MiB.

Como vemos, *heapy* nos ofrece mucha información respecto al uso de memoria de *Python*, por lo que surge como la herramienta ideal para analizar el uso de memoria de nuestros programas.

Tanto *psutil* como *Guppy3* son herramientas muy útiles si nuestro objetivo es simplemente ver la memoria total usada en cada momento, o ver en profundidad el estado del *heap*. Sin embargo, ambos módulos carecen de la funcionalidad esencial para la medición y limitación de memoria que buscamos en este trabajo:

- En primer lugar, ninguno de ellos actúa como monitor continuo de la memoria, simplemente responden a llamadas. Dado que nuestro objetivo final es el de obtener la máxima memoria usada a lo largo de toda de la ejecución, esta situación nos obligaría a tener un proceso paralelo que estuviese continuamente interrumpiendo la ejecución del

programa a analizar para obtener el pico máximo, aunque ni siquiera esto nos aseguraría el obtener el pico real.

- Por otra parte, ninguno de ellos nos ofrece la posibilidad de limitar el uso de memoria, únicamente de medirla, por lo que claramente vamos a necesitar al menos otro tipo de solución para implementar una limitación a la memoria máxima usada.

2.8. Limitación del consumo de memoria

Al contrario que para medir la memoria, donde todas las soluciones que hemos explorado eran multiplataforma, hasta donde sabemos no existen herramientas que nos permitan limitar la memoria máxima que puede utilizar un programa *Python*. Dado que el ámbito final del proyecto es un entorno *Linux*, en esta sección vamos a centrarnos en esta plataforma.

Para la parte de medir la memoria máxima usada, tenemos la llamada al sistema `getrusage`(Mitchell et al., 2001), que nos devuelve una estructura con diferentes métricas del proceso indicado en la llamada. Entre estas, tenemos `ru_maxrss`, que nos indica en KiB la máxima memoria usada por el proceso indicado, lo que coincide exactamente con la funcionalidad que estábamos buscando.

Esta función, además de ser usable desde un programa en C, está implementada en la misma librería estándar de *Python*, en el módulo *resource*(Van Rossum y Drake, 1995) el cual es exclusivo de *Linux*. Con esta información, podemos escribir una función sencilla:

```
from resource import getrusage, RUSAGE_SELF

def get_used_memory():
    mem = getrusage(RUSAGE_SELF).ru_maxrss * 1024
    print("Peak memory (MiB): {:.2f}".format(mem / 1024**2))
    return mem

get_used_memory()
l = [getX(-10) for a in range(1000000)]
get_used_memory()
l = [getX(10) for a in range(1000000)]
get_used_memory()
l = []
get_used_memory()
```

En ella, indicamos a la función `getrusage` que obtenga la máxima memoria usada por el proceso actual, *RUSAGE_SELF*, en bytes. Veamos la salida que produce con las listas ya vistas:

```
$ python maxMemory.py
Peak memory (MiB): 8.12
```

```
Peak memory (MiB): 47.01
Peak memory (MiB): 54.63
Peak memory (MiB): 54.63
```

Como se puede observar, tanto la primera como la segunda consulta son muy similares a los módulos utilizados en la sección anterior. Por el contrario, en la tercera consulta, la de la lista que no ocupa tamaño por estar en la caché de enteros, vemos que la memoria utilizada asciende en unos 8 MiB. Este aumento de memoria es debido a que el recolector de basura no actúa cuando la lista es reescrita, sino que simplemente se le asigna una nueva dirección y se retiene en memoria. La última lectura es realizada después de asignar la lista `l` a una lista vacía, que ya hemos comprobado que hace actuar al recolector y aún así mantenemos la misma memoria que en la anterior, esperable teniendo en cuenta que estamos midiendo el *pico* de memoria, no la que actualmente reside en memoria.

Si en vez de reasignar la lista, hubiéramos asignado la lista a una vacía

```
l = []
```

obtendríamos los siguientes resultados:

```
$ python maxMemory.py
Peak memory (MiB): 8.20
Peak memory (MiB): 46.98
Peak memory (MiB): 46.98
Peak memory (MiB): 46.98
```

Esta vez, el recolector de basura ha actuado, por lo que la memoria de la tercera lectura es la misma que la segunda. Recordemos que, en contraste a las mediciones obtenidas con los módulos anteriores, en esta ocasión estamos obteniendo la *máxima* memoria usada, por lo que obtener la misma medida es correcto.

Para la parte de *limitar* la máxima memoria utilizable por un proceso, tenemos la llamada al sistema `setrlimit`(Mitchell et al., 2001). En este caso, a la función le tenemos que indicar que recurso queremos limitar y la cantidad usando una estructura `rlimit`. El recurso a limitar es la memoria usada por los datos del programa (sin contar la inicialización), correspondiente al recurso *RLIMIT_DATA*. La siguiente función utiliza esta llamada para limitar la memoria del proceso actual a la cantidad indicada por *limit* en MiB:

```
def limit_memory(limit):
    rsrc = resource.RLIMIT_DATA
    soft, hard = resource.getrlimit(rsrc)
    resource.setrlimit(rsrc, (limit * 1024**2, hard))
    return
limit_memory(50)
```

Si ejecutamos con las listas de siempre, que hemos visto que ocupan un máximo de 47 MiB, podemos esperar que termine con normalidad:

```
$ python limitMemory.py
Peak memory (MiB): 7.99
Peak memory (MiB): 46.93
Peak memory (MiB): 46.93
```

mientras que si limitamos la memoria máxima a 40 MiB, lo esperable es que el proceso sea interrumpido por el sistema operativo:

```
$ python limitMemory.py
Peak memory (MiB): 8.07
Traceback (most recent call last):
  File "/media/sf_TFG/TrabajoActual/limitMemory.py", line 26, in <module>
    l = [getX(-10) for a in range(1000000)]
  File "/media/sf_TFG/TrabajoActual/limitMemory.py", line 26, in <listcomp>
    l = [getX(-10) for a in range(1000000)]
MemoryError
```

Con estas dos últimas funciones, `getrusage` y `setrlimit`, contamos con las herramientas necesarias para desarrollar la parte de medición y limitación de memoria del entorno de ejecución de programas en *Python*.

Capítulo 3

Ejecución Segura

Ser capaz de ejecutar cualquier tipo de programa sin tener que preocuparse por intenciones maliciosas de sus usuarios es posiblemente la parte más importante del ejecutor de un juez en línea.

Para conseguirlo, debemos asegurarnos que nuestro entorno es lo suficientemente versátil como para permitir la ejecución de aquellos programas que cumplan con la normativa establecida en el juez, y lo suficientemente robusto como para no verse afectado, detectar y actuar en consecuencia ante cualquier código que consideremos malintencionado.

En este capítulo abordaremos la tarea de proteger el entorno de ejecución ante todo tipo de código enviado por los usuarios.

3.1. Restricción de llamadas al sistema

El primer mecanismo que vamos a utilizar para la protección del entorno es el bloqueo de llamadas al sistema. Recordemos, que la plataforma objetivo de este proyecto es un entorno *Linux* y que la construcción de un entorno de ejecución segura es completamente dependiente de la plataforma donde se ejecutará.

> Una llamada al sistema es un método o función que puede invocar un proceso para solicitar un cierto servicio al sistema operativo.

Las llamadas al sistema en *Linux* son el mecanismo por el cuál el usuario se comunica con el núcleo (o *kernel*) del sistema operativo. Entre muchas otras, `read` y `write` son llamadas al sistema que en este caso nos permiten leer y escribir en ficheros abiertos (por otra llamada al sistema, `open`). La lista completa de llamadas al sistema se encuentra en la segunda sección del manual de *Linux*[1].

[1] `https://man7.org/linux/man-pages/dir_section_2.html`

Resulta fácil ver el cómo, si bloqueamos por ejemplo la llamada al sistema `open`, cualquier intento de abrir un fichero será interrumpido por el sistema operativo. Con esta misma filosofía, vamos a adentrarnos en el mundo de la detección e interrupción de llamadas al sistema.

Para lograr esta tarea necesitaremos:

- Una forma de detectar qué llamadas al sistema realiza un programa
- Una forma de bloquear determinadas llamadas al sistema

La primera es necesaria para obtener qué llamadas al sistema son usadas por cada programa, mientras que la segunda nos permitirá acabar con aquel proceso que utilice una llamada prohibida.

3.2. Llamadas al sistema realizadas por un programa

Para ver qué llamadas al sistema realiza un programa, Linux cuenta con una herramienta llamada *strace*[1]. Si consultamos el manual (`man strace`)[2], *strace* ejecuta el comando especificado hasta su finalización, interceptando y registrando las llamadas al sistema que el proceso va realizando, así como las señales que recibe. No solo obtiene las propias llamadas, sino también los argumentos con los que se llaman y sus valores de retorno.

El siguiente fragmento de terminal ilustra la ejecución de esta herramienta con el comando `ls`:

```
$ strace ls
execve("/usr/bin/ls", ["ls"], 0x7ffebc4d7180 /* 51 vars */) = 0
brk(NULL)                               = 0x555757b16000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd9956f010) =
                              -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)  =
                       -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
[... otras llamadas no mostradas]
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Como podemos observar, hasta el comando más simple genera una enorme cantidad de llamadas (en el ejemplo no se muestra toda la salida, dado que `ls` da lugar a *62* llamadas). La opción `-c` de *strace* nos organiza la información de las llamadas ejecutadas en forma de tabla:

[1] Las ejecuciones se han realizado sobre una máquina ejecutando la distribución *Arch Linux*.

[2] `https://man7.org/linux/man-pages/man1/strace.1.html`

```
$ strace -c ls
exe.c in  out  out2  p  p.py  py_exe.py test.py
% time     seconds usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 19,55    0,000578          44        13           mmap
 17,75    0,000525         105         5           openat
 [... otras llamadas no mostradas]
  0,71    0,000021          21         1           write
  0,37    0,000011           5         2           getdents64
------ ----------- ----------- --------- --------- ----------------
100,00    0,002957          49        60         6 total
```

Estas tablas nos serán de gran utilidad posteriormente, cuándo tengamos que obtener las llamadas al sistema que debemos permitir ejecutar a los programas y aquellas que debemos bloquear.

3.2.1. Bloquear llamadas al sistema

Saber qué llamadas tenemos que permitir y cuáles tenemos que bloquear no es útil si no tenemos una forma efectiva de realizar estos bloqueos. Afortunadamente existe una llamada al sistema de Linux que nos permite hacer justamente esto, bloquear las llamadas al sistema que deseemos, `seccomp`(Corbet, 2009):

```
int seccomp(unsigned int operation,
                  unsigned int flags, void *args);
```

Sin embargo, esta vez no vamos a utilizar directamente esta llamada al sistema para realizar esta tarea. Vamos a optar por hacer uso de una librería de C para que haga por nosotros la interacción con el sistema. Concretamente, la librería *libseccomp*[1], de la cuál vamos a dar uso de varias funciones.

En primer lugar, la función que inicializa el filtro:

```
scmp_filter_ctx seccomp_init(uint32_t def_action);
```

Esta función inicializa el estado de un filtro de *seccomp*, lo prepara para su uso y pone *def_ action* como acción por defecto. Esto es, qué hacer cuándo el programa hace uso de alguna llamada al sistema que hemos prohibido.

En nuestro caso, la acción por defecto que más nos interesa es la de finalizar el proceso enviándole una señal SIGSYS, SCMP_ACT_KILL. Con este parámetro, forzamos a los programas a finalizar su ejecución si realizan alguna llamada al sistema que no esté permitida *explícitamente*. Por ello, en

[1] `https://github.com/seccomp/libseccomp`

lugar de determinar qué llamadas vamos a bloquear, tenemos que determinar aquellas que vamos a *permitir*.

Con el filtro inicializado, la siguiente función nos permite añadir llamadas al filtro, en nuestro caso con el objetivo de permitir su utilización:

```
int seccomp_rule_add(scmp_filter_ctx ctx, uint32_t action,
                     int syscall, unsigned int arg_cnt, ...);
```

Como primer argumento le pasamos el filtro que obtuvimos con la función anterior. En el segundo, le indicamos cuál será la acción a realizar en caso de que esta llamada se utilice. Dado que por defecto, todas las llamadas están bloqueadas, debemos indicarle cuáles son las que queremos permitir, con el parámetro SCMP_ACT_ALLOW. En el siguiente argumento, *syscall*, simplemente tenemos que señalar la llamada al sistema que permitimos ejecutar.

Después de estos tres argumentos, podemos añadir un número indeterminado de condiciones a las reglas. La funcionalidad más destacable es la de indicar el valor de los argumentos de una cierta llamada. Por ejemplo, para permitir que un programa pueda escribir en la salida estándar, pero que no pueda hacerlo en cualquier otro fichero, utilizamos estos argumentos extras.

Con todas las reglas definidas, lo único que nos falta es cargar el filtro para empezar a bloquear llamadas:

```
int seccomp_load(scmp_filter_ctx ctx);
```

Le pasamos como argumento el filtro creado por `seccomp_init()` y a partir de ese momento, todas las llamadas al sistema que no formen parte de la lista de permitidas resultará en la terminación del programa.

Con estas tres funciones a nuestra disposición, tenemos todo lo necesario para realizar la implementación de un entorno de ejecución de *Python* seguro, siempre y cuando permitamos las llamadas mínimas que utiliza el intérprete al iniciarse y aquellas empleadas en la ejecución de un programa estándar.

3.2.2. Llamadas a permitir en Python

Obtener las llamadas que realize el intérprete de *Python* para inicializar su entorno es tan simple como ejecutar un programa *Python* vacío junto a la herramienta *strace*:

```
$ strace -c python -c ""
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 24,62    0,004492          27       165        13 newfstatat
 23,84    0,004350          83        52         4 openat
 11,65    0,002125          31        68           rt_sigaction
 10,49    0,001914          22        85           read
  9,89    0,001804          27        65         3 lseek
  8,14    0,001486          39        38        32 ioctl
  5,94    0,001083          21        51           close
  1,58    0,000289          20        14           getdents64
  1,52    0,000278          27        10           brk
  1,23    0,000225           5        39           mmap
  0,65    0,000119          39         3           dup
  0,22    0,000041          41         1           sysinfo
  0,21    0,000039          39         1           fcntl
  0,00    0,000000           0        10           mprotect
  0,00    0,000000           0         1           munmap
  0,00    0,000000           0         1           rt_sigprocmask
  0,00    0,000000           0         6           pread64
  0,00    0,000000           0         1         1 access
  0,00    0,000000           0         1           execve
  0,00    0,000000           0         3         1 readlink
  0,00    0,000000           0         1           getuid
  0,00    0,000000           0         1           getgid
  0,00    0,000000           0         1           geteuid
  0,00    0,000000           0         1           getegid
  0,00    0,000000           0         2         1 arch_prctl
  0,00    0,000000           0         1           futex
  0,00    0,000000           0         1           set_tid_address
  0,00    0,000000           0         1           set_robust_list
  0,00    0,000000           0         1           prlimit64
  0,00    0,000000           0         1           getrandom
------ ----------- ----------- --------- --------- ----------------
100,00    0,018245          29       626        55 total
```

En este comando, indicamos a *strace* que nos muestre un resumen de las llamadas (opción `-c`) y que lo haga sobre *Python* ejecutando el *script* que se le pasa por parámetro (curiosamente, opción `-c` también), en este caso, un programa vacío. Si creamos un programa C que permita la ejecución solo de estas llamadas, debería funcionar sin problema.

Esto no quiere decir que cualquier programa *Python* cuya ejecución queremos permitir no vaya a utilizar llamadas diferentes a las obtenidas. Al mismo tiempo, no debemos pensar que permitir todas estas llamadas va a resultar en un entorno lo suficientemente seguro como para ejecutar programas sin cuidado. Si somos más observadores, entre otras, vemos que se utilizan las llamadas `openat`, `read` y `write`, con las cuáles un programa sería capaz de abrir, leer y escribir en cualquier fichero del entorno de ejecución.

Es ahora cuándo recordamos los argumentos extras de los que dispone la función de añadir reglas, `seccomp_rule_add`. Con estos, vamos a permitir que se abran, lean y escriban si cumplen las siguientes condiciones:

- `openat`: no se permite escribir ni crear ficheros, es decir, los *flags* de la llamada no pueden ser O_WRONLY ni O_CREAT. Esto deja la posibilidad de abrir ficheros para su lectura, necesaria para que el intérprete inicie su ejecución.

- `read` y `write`: solo se puede leer y escribir sobre un fichero si estos son la entrada o salida estándar, y la salida de error. Es necesario poder leer de entrada estándar y escribir a salida estándar para realizar los problemas, pero no permitimos que se realicen sobre ningún otro fichero.

3.2.3. Implementación

Una vez hemos obtenido todo lo necesario, las funciones a utilizar y cómo usarlas, y las llamadas a permitir, pasemos a realizar la implementación. En primer lugar, definimos una función que inicializa el filtro y le añade todas las llamadas que debemos permitir como mínimo:

```
scmp_filter_ctx setup_seccomp(){
    scmp_filter_ctx ctx = seccomp_init(SCMP_ACT_KILL);

    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(access), 0);
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(arch_prctl), 0);
    ... ...
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(sysinfo), 0);
    //Allow only with these arguments
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(openat), 1,
           SCMP_A2(SCMP_CMP_EQ, O_RDONLY));
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(openat), 1,
           SCMP_A2(SCMP_CMP_EQ, O_RDONLY|O_CLOEXEC));
    seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(openat), 1,
           SCMP_A2(SCMP_CMP_EQ,
                  O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC));
    //Allow only for stdin, stdout y stderr
    for (int i = 0; i < 3; i++) {
        seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 1,
            SCMP_A0(SCMP_CMP_EQ, i));
        seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(read), 1,
            SCMP_A0(SCMP_CMP_EQ, i));
    }
    return ctx;
}
```

Programa 3.1: Inicializar el filtro

Llamamos a `init()` indicándole que la acción por defecto sea terminar el programa y comenzamos a añadir reglas. Como comentábamos anteriormente, la función `openat` solo se permite si no va a escribir o crear un fichero

nuevo, y las funciones `read` y `write` solo para la entrada y salida estándar, y la salida de error, que corresponden a los descriptores de ficheros[1] 0, 1 y 2.

Lo único que nos falta es cargar este filtro sobre un programa y probar que funciona correctamente:

```
//compile with -lseccomp
scmp_filter_ctx ctx = setup_seccomp();
char *args[] = { "/usr/bin/python", argv[1], 0};

pid_t childPid = fork();
if(childPid == 0){
    //Proceso hijo
    if (seccomp_load(ctx) != 0) {
        printf("ERROR: Couldn't load execution filters.");
        exit(-1);
    }
    execvp(args[0], (char **const) &args);
}
else{
    //Proceso padre
    int returnStatus;
    waitpid(childPid, &returnStatus, 0);
    printf("Returned value: %d\n", returnStatus);
}
```

Programa 3.2: Cargar el filtro y ejecutar un programa

En orden, llamamos a la función que inicializa el filtro y creamos los argumentos para ejecutar el programa, en este caso vamos a ejecutar programa en *Python*. Hacemos un `fork`, donde el hijo ejecutará el programa y el padre esperará hasta que el hijo termine.

El hijo, antes de ejecutar el programa, carga el filtro con la función `seccomp_load`. Las reglas establecidas son heredadas por el hilo de ejecución y los hijos, por eso el programa ejecutado con `execvp` estará bajo esas mismas reglas.

Por último y para asegurarnos de su correcto funcionamiento, vamos a ejecutar una serie de programas en *Python* que deberían hacer saltar las reglas:

Python	Resultado

`#Print ok, should work fine`
`print("ok")`
	`$ ./exe ok.py`
	`ok`
	`Returned value: 0`

`#Execute ls, should crash` `os.execve('/bin/ls', ["ls"], {})`	`$ ./exe execve.py` `Returned value: 159`

[1] https://en.wikipedia.org/wiki/File_descriptor

`#Fork the program, crash` `f = os.fork()`	`$ ./exe chdir.py` `Returned value: 159`

`#Open and read a file`
`f = open("e.in", "r")`
`for l in f:`
`  print(l)`
	`$ ./exe read.py`
	`soy e.in`
	`Returned value: 0`

`#Open and write to a file`
`f = open("e.in", "w")`
`  print("hey", file=f)`
	`$ ./exe write.py`
	`Returned value: 159`

El resultado no es más que el valor de salida capturado por el padre, que se quedó esperando la finalización del hijo. Si el programa ha terminado correctamente, obtendremos un 0, como ocurre en el primer programa. En caso contrario, obtendremos un valor diferente a 0, como los 159 que hemos obtenido en los siguientes programas.

Tanto `execve`, como `fork`, como el último `open` son finalizados de forma forzosa, ya que realizan llamadas que hemos bloqueado. El primer `open` vemos que solo abre el fichero para lectura, por lo que es capaz de leer el contenido del fichero y finalizar sin problema. Si por el contrario, en el segundo argumento donde se indican los *flags*, hubiésemos indicado que, si no existe el fichero se cree (en *Python* con '+'), esto se traduce al *flag* O_CREAT, el cuál está bloqueado, lo que finalizaría su ejecución forzosamente.

Un programa que lo único que puede hacer es leer ficheros, no escribir en ellos o crear nuevos, no es capaz de causar mucho daño al entorno. Sin embargo, queremos descubrir si existe alguna manera de ejecutar programas en *Python* con limitaciones mayores a las que nos permite el ejecutor desarrollado en esta sección.

Actualmente estamos limitados por las necesidades de carga del intérprete. Si no permitimos la ejecución de alguna de las llamadas que este realiza, no podremos ejecutar ningún programa. La idea entonces es, cargamos el intérprete de *Python* sin limitar las llamadas que puede realizar y luego establecemos las reglas necesarias solo para los programas, que podrán ser más restrictivas que para el intérprete.

Para llevar a cabo esta tarea, no restringiremos las llamadas desde el programa en C, sino que lo haremos en un programa *Python* intermedio que establecerá las reglas permitidas y luego se encargará de ejecutar el programa final. De esta forma, cuando establecemos las restricciones el intérprete ya ha sido cargado. Por supuesto, dependemos de si existe alguna forma de limitar las llamadas al igual que en C.

Afortunadamente, existe. ***Pyseccomp***[1] hace uso de la misma librería de C que hemos estado utilizando hasta ahora y su utilización, aunque confusa

[1] `https://github.com/cptpcrd/pyseccomp`

ante la falta de documentación, no es demasiado complicada.

El siguiente programa *Python* realiza una funcionalidad muy similar al ejecutor en C:

```
import sys, os
from pyseccomp import *

f = SyscallFilter(defaction=KILL)
f.add_rule(ALLOW, "exit_group")
f.add_rule(ALLOW, "rt_sigaction")
f.add_rule(ALLOW, "brk")
f.add_rule(ALLOW, "fstat")
f.add_rule(ALLOW, "ioctl")
f.add_rule(ALLOW, "lseek")
f.add_rule(ALLOW, "read")
f.add_rule(ALLOW, "close")
f.add_rule(ALLOW, "mmap")
f.add_rule(ALLOW, "write")

program = open(sys.argv[1]).read()
f.load()
exec(program)
```

Programa 3.13: Restricción en Python

Establece el filtro con la acción por defecto de terminar la ejecución y añade las reglas suficientes para ejecutar los programas. En este punto, en vez de cargar los filtros, abrimos y leemos el programa como *string* y luego cargamos el filtro. Esta forma de ejecución tan rara se debe a dos motivos:

1. Si hiciéramos primero la carga de las restricciones no podríamos abrir el fichero a ejecutar.

2. La función `exec()` de *Python* permite ejecutar programas en forma de *string*. Además, no realiza ninguna llamada al sistema, ya que el programa se encuentra en la memoria perteneciente a *Python* y a que no se utiliza un nuevo hilo de ejecución, sino que el intérprete continúa la ejecución como si fuera parte del propio programa.

Con este nuevo ejecutor, es evidente que no podemos abrir archivos de ninguna forma, ya que no permitimos la llamada `open` en ningún momento.

Como cierre de esta sección, queremos dejar claro que en ningún momento aseguramos que las llamadas permitidas en ambos ejecutores sean todas las necesarias para cualquier programa. De hecho, lo más seguro es que en un futuro esta lista crezca con nuevas llamadas realizadas por programas que sí deberían ser capaces de ejecutarse. Todo dependerá de los resultados de las pruebas están descritas en la sección de Validación.

3.3. Ejecución segura mediante *chroot*

La segunda medida de seguridad utilizada es la de ejecutar los programas dentro de un ***chroot***(Kamp y Watson, 2000; Friedl, 2002). Utilizándola vamos a crear un entorno dónde lo único que podemos llevar a cabo será la ejecución del intérprete de *Python*. Al mismo tiempo, la creación de este entorno, como veremos en la sección 3.3.1, nos va a permitir restringir los módulos de la librería estándar de *Python* que los usuarios podrán utilizar.

chroot es en el fondo una operación que cambia el directorio raíz[1] del sistema para un proceso y sus hijos. Un programa ejecutado dentro de este entorno no es capaz de acceder a ficheros fuera del árbol de directorios designado. Este tipo de entornos es comúnmente conocido como "jaula *chroot*".

Además de no permitir el acceso a ficheros, cualquier herramienta que queramos utilizar dentro de la jaula debe estar disponible dentro de ella. En un *chroot* vacío no podemos ni siquiera lanzar la propia terminal, ya que necesita el ejecutable `/bin/bash`.

En principio, en el entorno de *chroot* no necesitaríamos nada para la ejecución de programas *Python*, ya que el intérprete se encuentra fuera de este entorno y su ejecución puede continuar sin problema.

Como prueba de ello, tenemos el siguiente programa *Python*, en el que utilizamos la función `chroot` para cambiar el directorio raíz, luego cambiamos el directorio de trabajo actual con `chdir` e imprimimos el directorio actual antes y después de estas llamadas:

```
import os
print(f"Fuera de chroot: '{os.getcwd()}'")
os.chroot('.')
os.chdir('/')
print(f"Dentro de chroot: '{os.getcwd()}'")
```

```
$ sudo python chroot.py
Fuera de chroot: '/home/david/Desktop/TEST'
Dentro de chroot: '/'
```

Vemos que el directorio de trabajo se ha cambiado de forma correcta, pero también vemos que no tenemos ningún problema para obtener el directorio de trabajo una vez estamos dentro. Esto se debe a que, aunque no podamos utilizar ninguna herramienta habitual (incluyendo la terminal), sí que podemos realizar llamadas al sistema. Esto, como hemos estado viendo en la sección anterior, no va a resultar problemático dado que ya contamos con una solución para este problema.

Para comprobar que, efectivamente, no tenemos acceso a la terminal, podemos ejecutar el siguiente programa:

[1] `https://en.wikipedia.org/wiki/Root_directory`

```
import os
os.chroot('.')
os.chdir('/')
ret = os.system("ls")
print(ret >> 8)
```

```
$ sudo python chroot-ls.py
127
```

Ejecutamos `chroot` y cambiamos el espacio de trabajo, y ejecutamos un comando (en *Python* esto se realiza con `os.system()`). En la salida vemos el código de retorno del este comando, 127, cuyo significado corresponde con "comando no encontrado"[1].

3.3.1. Ejecutar *Python* dentro de *chroot*

El desarrollo de un entorno de *chroot* que nos permita ejecutar *Python* ha sido posible en gran medida a la guía *The Joy of chroot*[2].

Los pasos a seguir para crear un entorno *chroot* con todo lo necesario para ejecutar el intérprete de *Python* son los siguientes:

1. Tenemos que identificar y copiar recursivamente las dependencias del intérprete. Esto es, todas las librerías y ficheros de los que depende para iniciar su ejecución. De esto se encarga el *script* `recursive_ldd.sh`, que podemos encontrar en la página mencionada anteriormente.

2. El intérprete de *Python* no solo depende de las librerías que hemos copiado en el punto anterior, también abre y lee multitud de archivos durante su inicialización que no existen actualmente en el entorno. La forma en la que vamos a obtener esta serie de ficheros es con una herramienta que ya conocemos, *strace*. Simplemente ejecutamos `strace python '<'` y guardamos la salida en un fichero.

3. Para obtener los ficheros abiertos a partir de la traza, utilizaremos otro *script* de la guía, `strace_log_open_files.sh`. Este *script* recorre todas las llamadas de la traza y se queda únicamente con aquellas que sean `open` y que hayan resultado en éxito (se haya logrado abrir el fichero). El *script* ha sido modificado dado que *Python* no realiza llamadas `open`, sino `openat`.

[1] En *Unix*, `os.system()` de *Python* permite ejecutar comandos de terminal. El código de retorno es un número de 16 bits donde los 8 más significativos indican el valor de retorno del comando y los 8 menos significativos la señal que terminó el proceso. Por eso desplazamos el valor devuelto por la función 8 posiciones, para obtener este valor de retorno.

[2] `https://crossbowerbt.github.io/the_joy_of_chroot.html`

4. Sabiendo todos los ficheros que utiliza, únicamente nos falta el copiarlos sobre el entorno. Este tercer *script*, `copy_open_files.sh`, ha sido escrito por nosotros.

El resultado de seguir todos los pasos es el de un entorno capaz de ejecutar el intérprete de *Python*, perfecto para encerrarlo en un *chroot*. Como no queremos realizarlos manualmente cada vez que inicialicemos el entorno, se ha creado el siguiente *script*:

```
#!/bin/bash

set -e
#Get root folder
root="$1"
#Copy dependencies of python and
./recursive_ldd.sh "/usr/bin/python" "$root"
#Obtain the trace of python and all modules allowed
strace python -c "" 2>log
#From the trace obtains all files opened
./strace_log_open_files.sh log > opens
#Copy those files to the root folder
./copy_open_files.sh opens $root
#Delete the files created
rm log
rm opens
#Ejecute python inside chroot
chroot "$root" python
```

Programa 3.14: Crear entorno Python

Si lo ejecutamos y todo funciona correctamente, debería abrirse al final el intérprete de *Python* dentro del *chroot* creado:

```
$ sudo chroot root/ python
Python 3.9.1 (default, Dec 13 2020, 11:55:53)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Dado que solo hemos copiado las dependencias de *Python*, dentro del entorno seguimos sin poder ejecutar ninguna de las herramientas típicas, como puede ser un simple 'ls'.

Probando cosas en la consola de *Python* dentro del entorno, nos puede parecer que todo funciona correctamente (exceptuando claro el ejecutar comandos, abrir archivos, etc). Sin embargo, todo este tiempo nos estamos olvidando de probablemente la parte más importante de *Python* y a la que actualmente no tenemos acceso, su librería estándar.

Compuesta por diferentes módulos, estos no se encuentran disponibles directamente en el intérprete, sino en diferentes ficheros *Python*, uno por

cada módulo. Si tratamos de importar alguno de ellos, obtenemos el siguiente error:

```
>>> import math
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'math'
```

Sin acceso a la librería estándar no podemos hacer una gran parte de los problemas disponibles en ningún juez: todas las estructuras de datos, las funciones matemáticas, soporte para horas y fechas y muchas otras utilidades se encuentran en ella. Igualmente, existen varios módulos a los que **no** queremos dar acceso dentro del entorno, por ejemplo los dedicados a la concurrencia (como *threading* o *subprocess*), a protocolos de internet (como *webbrowser* o *socketserver*) o a las interfaces gráficas (*tkinter*).

La forma más sencilla de solo permitir algunos módulos requiere el realizar una criba sobre todos ellos y guardar una lista blanca de los que permitiremos. Una vez la tengamos, creamos un programa que reciba esa lista e importe los módulos de uno en uno de la siguiente forma:

```
for library in module_list:
    try:
        exec("import {module}".format(module=library))
    except Exception as e:
        print(e)
```

Programa 3.15: Modulos Python

Para tenerlos disponibles dentro del entorno de *chroot*, debemos copiar todos los ficheros que abre la ejecución de este programa. Esto lo conseguimos haciendo la traza no solo del intérprete de *Python*, sino de este ejecutando el programa que carga todos los módulos, modificando esta línea del *script*:

```
strace python -c "import modules" 2>log
```

Si ejecutamos de nuevo el *script* que crea el entorno completo con esta modificación, ya seremos capaces de acceder a los módulos que hemos permitido:

```
>>> import math
>>> math.sqrt(10)
3.1622776601683795
```

Capítulo 4

Limitación de Memoria

En el último apartado del *Estado del arte* exploramos las diferentes formas que tenemos a nuestra disposición para medir la memoria utilizada por un programa *Python* y también el cómo podíamos limitar su consumo máximo.

A lo largo de este capítulo vamos a profundizar en los mecanismos que utilizaremos en la implementación final, explorando las diferentes opciones que nos ofrecen y eligiendo las más adecuadas para este proyecto.

Además, examinaremos más detenidamente el uso de memoria que hace *Python* en diferentes situaciones y problemas, para demostrar su viabilidad no solo en la programación competitiva en general, sino en aquellos problemas con la limitación de memoria como un punto principal en su resolución.

4.1. Medición de memoria

Como vimos en el último apartado del *Estado del arte*, vamos a utilizar llamadas al sistema de *Linux* para obtener la memoria máxima utilizada por un proceso *Python*. Concretamente, utilizaremos `getrusage`, cuya documentación podemos encontrar en el manual de Linux y en (Mitchell et al., 2001).

```
int getrusage(int who, struct rusage *usage);
```

Como gran parte de las llamadas al sistema, `getrusage` devuelve 0 en caso de que éxito y -1 en cualquier otro caso, estableciendo la variable *errno* para indicar el error. La función recibe dos argumentos. En el primero de ellos, *who*, le indicamos de "quién" queremos obtener el uso de recursos. Este puede ser uno de los siguientes:

- RUSAGE_SELF: para obtener los recursos utilizamos por el proceso actual, incluyendo todos los hilos generados por el mismo.

- RUSAGE_CHILDREN: para solo obtener los recursos utilizados por todos los hijos creados que hayan finalizado su ejecución.
- RUSAGE_THREAD: para obtener los recursos utilizados por el hilo actual.

De entre todos ellos, RUSAGE_CHILDREN es el que más se acerca a lo que queremos, que es medir únicamente la memoria utilizada por el intérprete de *Python*, que será invocado desde el código en C como un proceso hijo. El segundo argumento es el verdadero retorno de la función, la estructura *rusage* con todos los siguientes usos de recursos:

```
struct rusage {
   struct timeval ru_utime; /* user CPU time used */
   struct timeval ru_stime; /* system CPU time used */
   long   ru_maxrss;       /* maximum resident set size */
   long   ru_ixrss;        /* integral shared memory size */
   long   ru_idrss;        /* integral unshared data size */
   long   ru_isrss;        /* integral unshared stack size */
   long   ru_minflt;       /* page reclaims (soft page faults) */
   long   ru_majflt;       /* page faults (hard page faults) */
   long   ru_nswap;        /* swaps */
   long   ru_inblock;      /* block input operations */
   long   ru_oublock;      /* block output operations */
   long   ru_msgsnd;       /* IPC messages sent */
   long   ru_msgrcv;       /* IPC messages received */
   long   ru_nsignals;     /* signals received */
   long   ru_nvcsw;        /* voluntary context switches */
   long   ru_nivcsw;       /* involuntary context switches */
};
```

De entre todos ellos usaremos únicamente `ru_maxrss`, a pesar de que `ru_ixss`, `ru_idrss` y `ru_isrss` estén relacionados con la memoria y que nos podrían resultar ser útiles, ya que actualmente son campos no usados en *Linux*.

Si leemos la descripción de este vemos que nos devuelve la memoria máxima que en algún momento perteneció al proceso y que fue utilizada de forma activa, en este caso significando que estuvo residiendo en *RAM*. La memoria devuelta está medida en kilobytes.

En la misma descripción, nos informa que en caso de utilizar RUSAGE_CHILDREN, el campo indicará el tamaño del hijo más grande, no de la suma de todos los hijos creados en el árbol de procesos. En nuestro caso, esta limitación no nos afectará, pues solo pretendemos medir la memoria utilizada por un hijo, que ejecutará a su vez el programa *Python* y seguirá siendo el mismo proceso, pero además porque *Python*, como hemos visto y

veremos más adelante, usa una cantidad de memoria mucho mayor que un programa en C con la misma funcionalidad, por lo que en cualquier caso, el campo registraría el uso de memoria de *Python*.

Conociendo más en detalle el funcionamiento de `getrusage`, vamos a codificar un pequeño programa que ejecuta un comando pasado por parámetro y muestre la memoria usada por este al acabar:

```c
#include <stdio.h>
#include <stdlib.h>
//For getrusage
#include <sys/time.h>
#include <sys/resource.h>

int main(int argc, char *argv[]) {
    system(argv[1]);
    struct rusage use;
    getrusage(RUSAGE_CHILDREN, &use);
    printf("Memory used: %0.2fMB\n", (float)(use.ru_maxrss) / 1024.0);
    return 0;
}
```

Programa 4.1: Uso de memoria de comando

Con este programa podemos medir la cantidad de memoria que usa cualquier otro, simplemente ejecutándolo y pasándole por parámetro cualquier comando:

- El comando `ls`:

  ```
  $ ./memuse "ls"
  empty.py hello_world.py memlimit.c mem.py memuse memuse.c p use
  Memory used: 3.52MB
  ```

- Un programa en *Python*:

  ```
  $ ./memuse "python hello_world.py"
  Hello world!
  Memory used: 7.66MB
  ```

- Incluso a sí mismo, si tenemos cuidado con los parámetros:

  ```
  $ ./memuse "./memuse \"\""
  Memory used: 3.58MB
  Memory used: 3.58MB
  ```

4.2. Limitación de memoria

De la misma forma utilizada para medir el uso de memoria, la limitación de esta la vamos a implementar utilizando otra llamada al sistema de *Linux*,

en este caso `setrlimit`(Mitchell et al., 2001).

```
int setrlimit(int resource, const struct rlimit *rlim);
```

De nuevo, tenemos otra función que devuelve 0 en caso de éxito y -1 en caso de error, estableciendo la variable *errno* con el error ocurrido.

La función recibe dos argumentos. En el primero, *resource*, indicamos a la función qué recurso queremos limitar. Entre los recursos que podemos limitar, se encuentran la máxima memoria virtual (RLIMIT_AS), el tiempo máximo de CPU que puede ocupar (RLIMIT_CPU), el número de archivos que puede crear (RLIMIT_FSIZE), y muchos otros que se pueden consultar en la documentación. De entre todos ellos nos interesan especialmente dos:

- RLIMIT_DATA: para limitar el máximo tamaño de segmento de datos del proceso, que incluye los datos inicializados y sin inicializar y el *heap*. El límite se establece en Bytes y es redondeado hacia abajo en base al tamaño de página del sistema.
- RLIMIT_STACK: para limitar el máximo tamaño de la pila del proceso. En *Python* es especialmente importante cambiar este límite debido a que por defecto está limitado a 8 MiB (8.388.608 Bytes). Estudiaremos el límite de recursión de *Python* más en profundidad en la sección 4.3.1.

El hecho de que ambos recursos se limiten por separado, pero que obtengamos su suma al medir la memoria máxima usada es ligeramente problemático. Teóricamente, si limitásemos un programa a usar un máximo de 50 MiB, tanto en datos como el pila, este sería capaz de utilizar hasta un máximo de 100 MiB, 50 de datos y 50 de pila, sin que el sistema operativo le envíe una señal forzándole a terminar. Este caso también lo estudiaremos con más detalle en los siguientes apartados.

Como segundo argumento, tenemos que indicarle la limitación que ponemos sobre *resource*, con una estructura *rlimit*:

```
struct rlimit {
   rlim_t rlim_cur; /* Soft limit */
   rlim_t rlim_max; /* Hard limit (ceiling for rlim_cur) */
};
```

Donde *rlim_cur* indica el verdadero límite que el sistema operativo impone sobre el proceso. Si se utiliza un valor superior a este, el programa será terminado de forma forzosa. Este valor, conocido como *soft limit*, puede ser modificado por un usuario sin privilegios (es decir, no hace falta ser *root* en *Linux*) desde 0 hasta el otro campo de la estructura, *rlim_hard*. Este actúa

como *hard limit* y solo puede ser modificado por un usuario *root*. Por ello, si queremos limitar un programa a un valor máximo de memoria, debemos modificar ambos campos, ya que tendremos permisos para hacerlo. Por otra parte, el usuario ejecutará sin privilegios, por lo que no podrá aumentar el límite permitido, solo disminuirlo, opción que no aportaría ventaja alguna.

Al igual que hicimos para la medición de memoria, vamos a escribir otro programa en C que ejecute un comando pasado por parámetro, limitando su memoria también pasada como parámetro. Además, para ver si el comando se ejecutó correctamente capturamos el resultado de ejecutar el comando y lo escribimos.

```c
#include <stdio.h>
#include <stdlib.h>
//For setrlimit
#include <sys/time.h>
#include <sys/resource.h>

void limit_memory(int MB){
    struct rlimit data_limit = {MB, MB};
    struct rlimit stack_limit = {MB , MB};
    setrlimit(RLIMIT_DATA, &data_limit);
    setrlimit(RLIMIT_STACK, &stack_limit);
}

int main(int argc, char *argv[]) {
    long limit = strtol(argv[1], NULL, 10);
    limit_memory(limit*1024*1024);
    int ret = system(argv[2]);
    printf("Returned: %d\n", ret);
    return 0;
}
```

Programa 4.2: Limitar memoria de un comando

Podemos utilizarlo para ejecutar diferentes programas, como por ejemplo:

```
$ ./memlimit 50 "python list_1000000.py"
Returned: 0
```

En este caso, ejecutamos un programa en *Python* que genera una lista de 1 millón de enteros, del 1 al 1.000.000. La lista ocupa, según los resultados que vimos en el capítulo 2, alrededor de 47 MiB, por lo que el programa debería ejecutarse correctamente, hecho que podemos comprobar con el valor de retorno, 0.

Si bajamos el límite a 40 MiB, obtenemos el siguiente resultado:

```
$ ./memlimit 40 "python list_1000000.py"
Traceback (most recent call last):
  File "/media/sf_TFG/MemoryUseC/list_1000000.py", line 1, in <module>
    l = [i for i in range(1000000)]
  File "/media/sf_TFG/MemoryUseC/list_1000000.py", line 1, in <listcomp>
    l = [i for i in range(1000000)]
MemoryError
Returned: 256
```

Como era esperable, el programa utiliza más memoria de la que le hemos permitido, por lo que el sistema operativo finaliza su ejecución y nos devuelve algo diferente a 0. En esta instancia además, el intérprete de *Python* nos indica la excepción encontrada, *MemoryError* e incluso nos dice el segmento de código en el que se ha producido esta excepción.

4.3. Uso de memoria de Python

Con las formas obtenidas para medir y limitar la memoria, tenemos las herramientas necesarias para evaluar los diferentes programas en estos aspectos. Al mismo tiempo, estas no sirven de nada si no conocemos cuánto tenemos que limitar la memoria en los diferentes tipos de problemas, o simplemente que *Python* sea incapaz de realizar tareas que otros lenguajes sí que pueden.

Por ejemplo, entrando de nuevo en los jueces en línea, en *¡Acepta el Reto!* la mayor parte de los problemas tienen un límite de memoria de 4096 KB, memoria de sobra para un programa en C/C++ que utiliza pocas variables, mientras que si nos fijamos en los datos obtenidos en los apartados de medición, vemos a *Python* utilizar un mínimo de alrededor de 8 MiB únicamente para iniciar su ejecución.

En este apartado vamos a investigar verdaderamente la viabilidad de *Python* como lenguaje en problemas donde la limitación de memoria severa es una parte importante del problema.

4.3.1. Límite de recursión

Una técnica muy utilizada en todo tipo de problemas es la recursión, funciones que se llaman a sí mismas una y otra vez.

Entre otros, tenemos el algoritmo de grafos *DFS*, *Depth First Search* o búsqueda en profundidad, utilizado para la búsqueda de componentes conexas, la detección de ciclos, detección de puentes y puntos de articulación, etc.

El límite de recursión de *Python* está limitado por defecto a 1000 llamadas. Este hecho es comprobable con el módulo `sys`(Van Rossum et al., 2000) de la librería estándar y el método `getrecursionlimit()`:

```
import sys
print(sys.getrecursionlimit())
```

Programa 4.3: get_rec_limit.py

```
$ python get_rec_limit.py
1000
```

O de forma empírica, con la siguiente función:

```
def rec(n):
    print(n)
    rec(n+1)
rec(1)
```

Programa 4.4: rec_limit.py

```
$ python rec_limit.py
...
995
996
[... errores no mostrados]
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Por suerte, este límite es artificial, impuesto por el mismo intérprete de *Python* para evitar *stack overflow* innecesarios y no es un límite del propio lenguaje, lo cuál queda reflejado al poder ser cambiado desde la misma librería estándar sin necesidad de privilegios de *root*:

```
import sys
sys.setrecursionlimit(2000)
print(sys.getrecursionlimit())
```

Programa 4.5: get_rec_limit2.py

```
$ python get_rec_limit2.py
2000

$ python rec_limit.py
...
1995
1996
[... errores no mostrados]
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Lo ideal para programación competitiva sería eliminar este límite por completo, o en su defecto asignar un valor absurdamente grande (1000000000) y despreocuparse de ello. Haciendo este estamos ante un escenario similar

al de otros lenguajes en donde el desbordamiento está relacionado con el consumo de memoria de la pila y no limitado por el intérprete:

```
import sys
sys.setrecursionlimit(1000000000)

def rec(n):
    print(n)
    rec(n+1)
rec(1)
```

Programa 4.6: rec_limit2.py

```
$ python rec_limit2.py
...
20134
20135
Segmentation fault (core dumped)
```

Ya no obtenemos más el error de *RecursionError*, debido a que únicamente salta cuando alcanzamos el límite autoimpuesto por el intérprete, sino que directamente obtenemos un *Segmentation fault*. Con errores tan concretos es fácil trabajar, aunque finalmente encontramos la respuesta con las mismas herramientas que limitamos nosotros mismos la memoria máxima, en caso de *Python* el módulo `resource`.

De forma análoga a limitar, usando `setrlimit`, existe la función que obtiene las limitaciones actuales de los recursos, `getrlimit`, la cuál empleamos para obtener el tamaño máximo de la pila por defecto en *Python*:

```
import resource
print(resource.getrlimit(resource.RLIMIT_STACK))
```

Programa 4.7: stack_limit.py

```
$ python stack_limit.py
(8388608, -1)
```

La razón del *Segmentation fault* es que el programa sobrepasa el límite de memoria que puede utilizar en la pila y el sistema operativo finaliza su ejecución. Esto además nos permite realizar cálculos acerca del sobrecoste de la recursión en este lenguaje: con una limitación de 8.388.608 bytes es capaz de realizar un total de 20.135 llamadas (este número varía ligeramente entre ejecuciones), por lo que *Python* utiliza aproximadamente 416 bytes por cada llamada recursiva.

Si hacemos este límite ilimitado, con RLIM_INFINITY, las llamadas recursivas continuarán hasta utilizar toda la memoria disponible del sistema:

```
import sys, resource
sys.setrecursionlimit(1000000000)
resource.setrlimit(resource.RLIMIT_STACK, [resource.RLIM_INFINITY,
    resource.RLIM_INFINITY])

def rec(n):
    print(n, resource.getrusage(RUSAGE_SELF).ru_maxrss / 1024)
    rec(n+1)
rec(1)
```

Programa 4.8: rec_limit3.py

```
$ python rec_limit3.py
[... salidas de las llamadas anteriores]
1510281 1304.70703125
```

Además de la profundidad, la función también nos muestra la memoria utilizada por el proceso hasta el momento Antes que saltase un error de ejecución, la última impresión en la consola correspondía a la llamada 1.510.281, con un consumo de memoria actual de más de 1.2 GiB.

Esta misma es la razón por la que en los apartados de limitación de memoria mencionábamos la importancia de limitar la memoria, no solo de los datos, sino también de la pila. El tamaño por defecto de la pila en *Python* es muy pequeño para ejecutar ciertos algoritmos recursivos sobre casos de prueba voluminosos, y con las limitaciones que ponemos en el capítulo de ejecución segura no hay nada que los usuarios puedan hacer para modificar este campo, por lo que recae sobre nosotros el hacerlo.

4.3.2. Uso de memoria en entrada estándar

En todos los problemas de cualquier juez en línea y concurso de programación, los datos de entrada se leen por entrada estándar (*stdin*) y se escribe la salida por salida estándar (*stdout*).

Un tipo de problemas de *¡Acepta el Reto!* son aquellos con entradas de muchos elementos en la misma línea, los cuáles deben leerse de forma individual si no queremos sobrepasar el límite de memoria del problema. Entre estos, tenemos por ejemplo el 248 - "Los premios de las tragaperras", con un 30 % de MLE, el 129 - "Marcadores de 7 segmentos" con un 27 % o el 544 - "Que no se atraganten" con un 24 %.

Esta tarea es muy simple y de hecho la más natural en los lenguajes ya disponibles en la página, con *scanf* de C, *cin* en C++ y *Scanner* de Java. El problema surge en *Python*, donde no existe una forma de leer elementos de uno en uno.

La forma estándar de leer elementos de uno a uno en *Python* consiste en leer la línea entera como *string*, dividir la lista por espacios utilizando

la función `split()` y recorrer esta lista, obteniendo cada elemento. Esto no produce ningún problema en otros jueces que admiten *Python*, donde el límite de memoria es muy laxo, pero sí que es importante en *¡Acepta el reto!* dado que el objetivo de los autores de esos problemas suele ser, precisamente, que los usuarios ideen una solución que no necesite almacenar la lista en memoria. Si se incrementa el límite para poder almacenar toda la entrada, el problema aceptaría esas soluciones que intentábamos impedir con la limitación. Un ejemplo de programa que realiza esta función sería el siguiente:

```
suma = 0
for l in sys.stdin:
    for k in l.split():
        suma += int(k)
print(suma)
```

Programa 4.9: input1.py

En este programa simplemente leemos todas las líneas de la entrada estándar y cada una de estas líneas la convertimos en lista con `split`. El programa escribe finalmente la suma de todos los elementos leídos, para comprobar que ha funcionado correctamente. Si lo ejecutamos con nuestro medidor de memoria (al que se le ha añadido la capacidad de medir el tiempo de ejecución) y le pasamos una entrada con 1.081.080 de elementos en la misma línea (número cuya elección comentaremos más adelante) obtenemos:

```
$ ./memuse "python input1.py" < 1081080.in
584366442660
Memory used: 90.39MB
Time used: 0.49s
```

Como comentábamos previamente, el programa usa un total de ¡90 MiB! simplemente leyendo la entrada. Esta métrica resultaría en MLE en los problemas comentados. Al mismo tiempo, si permitiéramos que *Python* pudiese utilizar tal cantidad de memoria, acabaríamos con parte de la dificultad del problema y haríamos que utilizar este lenguaje sea la opción superior al resto en este tipo de problemas.

Por estas razones, debemos encontrar una forma alternativa de leer la entrada sin utilizar de forma innecesaria tanta memoria. La única funcionalidad que nos permite leer algo que no sea la línea entera es la función `read()` a la cuál le podemos indicar el número de bytes que queremos leer de un objeto de archivo (*file object*, una abstracción de un fichero en *Python* sobre los que podemos leer o escribir). Dado que la entrada estándar también se puede interpretar como un *file object*, podemos hacer la siguiente función:

```
import sys
def readInt():
    c = sys.stdin.read(1)
    while(c == ' ' or c == '\n'):
        c = sys.stdin.read(1)
    val = int(c)
    while(True):
        c = sys.stdin.read(1)
        if c == ' ' or c == '\n' or not c.isdigit():
            break
        val = val * 10 + int(c)
    return val

n = readInt()
suma = 0
for i in range(n):
    k = readInt()
    suma += k
print(suma)
```

Programa 4.10: input2.py

La función es muy sencilla, va leyendo de la entrada byte a byte hasta que encuentra un dígito (en este caso solo nos preocupamos por leer números, la función se puede adaptar fácilmente para leer otro tipo de datos). Una vez lo hace, sigue leyendo mientras sean dígitos y creando el número leído, multiplicando lo actual y sumándole el nuevo dígito. Veamos su consumo de memoria y tiempo:

```
$ ./memuse "python input2.py" < 1081080.in
584366442660
Memory used: 8.12MB
Time  used: 4.26s
```

Respecto al uso de memoria, es básicamente lo que buscabamos, ya que el intérprete utiliza unos 8 MiB para iniciarse. El problema está en tiempo de ejecución, que es alrededor de 10 veces más lento que el original. Podemos reducir ligeramente este tiempo cambiando el método de obtener el entero. En vez de multiplicar por 10 y sumar, añadimos el dígito al *string* y solo realizamos una conversión, al devolver el número:

```
def readInt():
    ...
    val = c
    ...
        val += c
    return int(val)
```

Programa 4.11: input2_2.py

```
$ ./memuse "python input2_2.py" < 1081080.in
584366442660
Memory used: 7.93MB
Time  used: 3.15s
```

Esta modificación nos ahorra alrededor de un segundo, pero aún estamos lejos del tiempo original. Una vez tenemos ya idea de cómo ser eficientes en memoria, nuestros esfuerzos se van a dirigir en intentar optimizar todo lo posible la lectura, para acercarnos lo máximo posible a ese tiempo.

En la siguiente iteración, vamos a convertir la función en una clase *reader*. En esta, en vez de leer la entrada de byte en byte, vamos a tener un *buffer* del que iremos obteniendo los caracteres hasta agotarlo, momento en el que lo volveremos a rellenar:

```
class reader:
    def __init__(self, lenBuffer=1024):
        self.lenBuff = lenBuffer
        self.fill_buffer()

    def fill_buffer(self):
        self.head = 0
        self.buffer = sys.stdin.read(self.lenBuff)
        if self.buffer == '':
            self.buffer = ' '
        self.end = len(self.buffer)

    def getChar(self):
        c = self.buffer[self.head]
        self.head += 1
        if self.head == self.end:
            self.fill_buffer()
        return c

    def getInt(self):
        ret = ""
        char = self.getChar()
        while not ('0' <= char <= '9'):
            char = self.getChar()
        while '0' <= char <= '9':
            ret += char
            char = self.getChar()
        return int(ret)
```

Programa 4.12: input3.py

```
$ ./memuse "python input3.py" < 1081080.in
584366442660
Memory used: 7.90MB
Time  used: 5.32s
```

Como lo único que hemos obtenido es aumentar el tiempo de ejecución, necesitamos otra estrategia para la lectura. La idea del *buffer* parece ser interesante y vamos a intentar explotarla de otra forma. En lugar de crear los números a partir de caracteres, vamos a utilizar la misma función que utiliza el original, `split`. Con esto, obtendremos una lista de *string* a partir del *buffer* de forma, esperemos, más eficiente y sin consumir demasiada memoria, ya que el *buffer* es relativamente pequeño:

```
class reader:
    def __init__(self, lenBuffer=1024):
        self.lenBuff = lenBuffer
        self.bufferList = []
        self.fill_buffer()

    def fill_buffer(self):
        self.head = 0
        self.buffer = sys.stdin.read(self.lenBuff)
        self.bufferList = self.buffer.split()
        if self.buffer == '': #EOF
            self.buffer = ' '
        self.end = len(self.bufferList)

    def getInt(self):
        ret = int(self.bufferList[self.head])
        self.head += 1
        if self.head == self.end:
            self.fill_buffer()
        return ret
```

Programa 4.13: input4.py

```
$ ./memuse "python input4.py" < 1081080.in
575638221219
Memory used: 8.11MB
Time  used: 1.07s
```

Esta modificación no solo hace el código más simple, sino que lo hace mucho más rápido, acercándonos a los tiempos de lectura estándares. Sin embargo, y por primera vez, el resultado que hemos obtenido de la suma es incorrecto. Esto se debe a "cortes" producidos por leer el *buffer* de longitud constante, donde alguno de los números ha sido leído parcialmente en un *buffer* y a la otra parte le ha ocurrido lo mismo.

Como ilustración de este error, si el tamaño del *buffer* fuera de 4 bytes y tenemos que leer el número "12345", el lector actual leería los números "1234" y "5", cortando el número original en dos.

Para solucionar este error, debemos controlar el primer y último caracter leído en el *buffer*. Si no es un caracter ignorable (espacios, tabuladores y saltos de línea), guardamos el último elemento de la lista leída en una variable

(*last*) e indicamos que la última lectura tuvo un elemento a medias. En el otro extremo, comprobamos si la última lectura tuvo un elemento a medias. Si es así, existen dos opciones:

- El elemento fue cortado en la lectura, por lo que tenemos que juntar el final de la lectura anterior con el primer elemento de esta.
- El elemento no fue cortado, sino que su último dígito fue exactamente el último carácter leído, por lo que lo insertamos íntegramente al principio de la lista de lectura.

Además de esta corrección, hemos intentado acelerar la lectura, evitando usar índices sobre la lista y usando por el contrario iteradores, dado que en *Python* tienden a ser más eficientes. Sin más comentarios, la siguiente iteración del lector:

```
class reader:
    def __init__(self, lenBuffer=4096):
        self.lenBuff = lenBuffer
        self.hasLeft = False
        self.bufferList = []
        self.fill_buffer()

    def fill_buffer(self):
        self.buffer = sys.stdin.read(self.lenBuff)
        self.bufferList = self.buffer.split()
        self.it = iter(self.bufferList)
        if self.buffer == '': #EOF
            self.buffer = ' '
        #First element
        if self.hasLeft:
            self.hasLeft = False
            #The element was split between reads
            if self.buffer[0] not in {' ', '\n', '\t'}:
                self.bufferList[0] = self.last + self.bufferList[0]
            else:
                self.bufferList.insert(0, self.last)
        #Last element
        if self.buffer[-1] not in {' ', '\n', '\t'}:
            self.hasLeft = True
            self.last = self.bufferList.pop()

    def getInt(self):
        try: ret = int(next(self.it))
        except StopIteration:
            self.fill_buffer()
            try: ret = int(next(self.it))
            except StopIteration:
                raise EOFError('EOF reached')
```

```
        return ret
```

Programa 4.14: input5.py

```
$ ./memuse "python input5.py" < 1081080.in
584366442660
Memory used: 8.28MB
Time  used: 0.76s
```

No solo hemos corregido el error en la entrada, sino que hemos hecho el programa aún más rápido con el cambio a iteradores. La diferencia de tiempos con la entrada estándar es la menor de todas las iteraciones hasta el momento, siendo unos 0.25s más lento leyendo alrededor de 1 millón de enteros.

Como conclusión de esta subsección, vamos a comparar la entrada que sería *estándar* en *Python*, *input1.py*, con nuestra propuesta de entrada eficiente, *input5.py* y vamos también a añadir a la comparación a C++. Este último realiza la lectura con la configuración típica para problemas de programación competitiva, utilizando `cin` para leer, desincronizando los flujos de C++ con los de C con `std::ios::sync_with_stdio(false)`[1] y desligando los operadores de entrada (`cin`) y salida (`cout`) con `cin.tie(nullptr)`[2].

En la primera gráfica 4.1, comparamos el tiempo de lectura de los tres programas de 1.081.080 enteros en base al número de columnas (número de elementos por fila). Así, con 1 columna tendremos un elemento por fila, 2 columnas 2 elementos por fila, etc. Por mera curiosidad matemática, el número 1.081.080 fue elegido debido a que es el *número altamente compuesto*[3] más cercano a 1 millón, lo que nos permite distribuir los elementos en 256 configuraciones (su número de divisores).

Hasta la generación de estas gráficas, asumíamos que la nueva entrada era más lenta que la entrada estándar de *Python* y que nuestro objetivo era acercarnos a su velocidad. Esta gráfica nos muestra que nuestra propuesta es más rápida siempre y cuando cada línea tenga menos de 8 elementos por fila, en donde son prácticamente iguales. A partir de este valor, la entrada original es ligeramente más rápida.

En la siguiente gráfica 4.2 veremos el consumo de memoria de los tres programas en la lectura de los mismos ficheros.

Como hemos ido viendo en las sucesivas iteraciones de nuestra entrada, la memoria utilizada se mantiene constante a los largo de las diferentes entradas gracias al *buffer* de tamaño constante que utiliza, mientras que la entrada original, al leer la fila entera, ve su uso de memoria incrementando en base al número de columnas.

[1] `https://en.cppreference.com/w/cpp/io/ios_base/sync_with_stdio`
[2] `https://en.cppreference.com/w/cpp/io/basic_ios/tie`
[3] `https://en.wikipedia.org/wiki/Highly_composite_number`

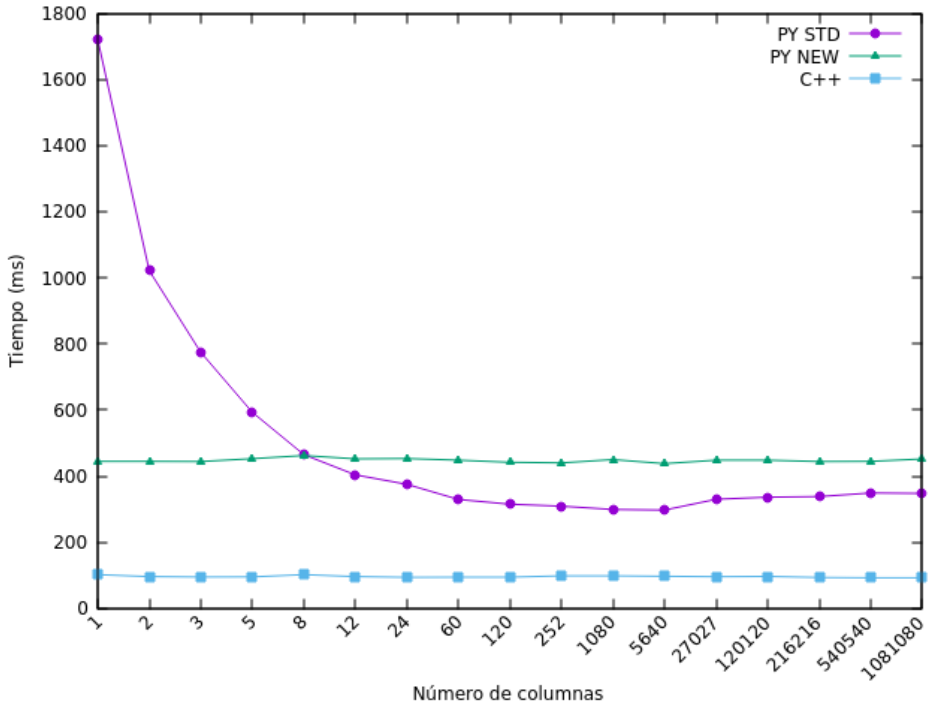

Figura 4.1: Tiempos de lectura

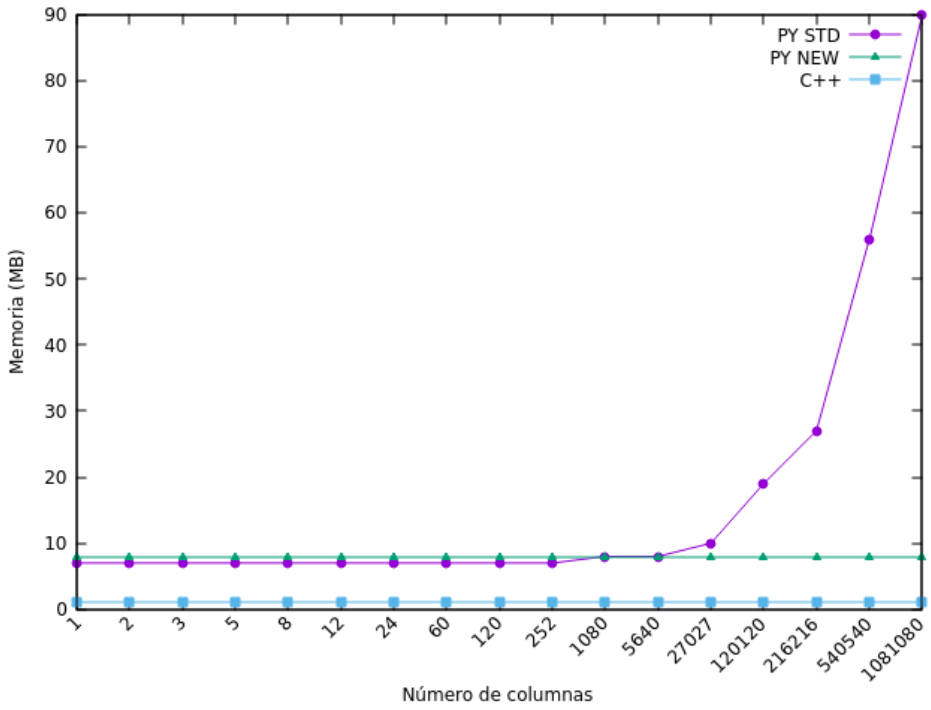

Figura 4.2: Uso de memoria de lectura

Estos resultados nos muestran que una lectura eficiente tanto en memoria como en tiempo es posible en *Python*. Claro está, los usuarios que se enfrenten a problemas de este tipo utilizándolo deben ser conscientes de que leer la entrada como están acostumbrados va a resultar en un MLE y que tienen que introducir un método alternativo para la lectura, como el definido en esta sección.

Capítulo 5

Implementación

En este capítulo nos vamos a dedicar a juntar todo lo que hemos desarrollado en los capítulos anteriores en un solo programa C.

Este programa servirá como ejecutor de programas escritos en *Python*, sobre los cuáles será capaz de establecer diferentes limitaciones. El ejecutor a su vez deberá asegurar nuestro entorno frente a programas malintencionados al mismo tiempo que permitirá la ejecución de programas que no lo sean.

Dado que el ejecutor está formado por las diferentes construcciones elaboradas anteriormente en este proyecto, así como algunas nuevas para alcanzar la funcionalidad deseada de este, su análisis se ha dividido en diferentes secciones, donde se explica en más detalle su funcionamiento.

5.1. Argumentos del programa

Para indicar al programa todos los parámetros con los que deseamos ejecutar un programa *Python* vamos a pasarle argumentos a través de la consola de comandos. Un ejemplo de ejecución es el que vemos a continuación:

```
$ ./exe -t 1000 -m 15000 -i f.in -o f.out f.py
```

En este caso, indicamos al programa (*exe*) que establezca el límite de tiempo en 1000 (medido en milisegundos), el límite de memoria en 15000 (en kibibytes), que la entrada del programa está en el fichero *f.in* y debe escribir sobre el fichero *f.out*. Y finalmente, que todo esto aplique sobre la ejecución del programa *f.py*.

El análisis de argumentos del programa trabaja sin problemas frente a argumentos desordenados y a la posición del programa a ejecutar. El siguiente comando es igual de válido que el anterior:

```
$ ./exe -i f.in -t 1000 f.py -o f.out -m 15000
```

Para ver todas las opciones que nos ofrece el programa, tenemos a nuestra disposición el argumento `-h`:

```
$ ./exe -h
Usage: sudo ./exe [OPTIONS] PROGRAM
Please, execute as super user
Run program with restrictions
  -t, --time_limit=TIME(ms)     set maximum CPU time to
                                TIME milliseconds
  -m, --memory_limit=MEMORY(kB) set maximum MEMORY use in kibibytes
  -i, --input_file=FILE         set the input file
  -o, --output_file=FILE        set the output file
  -c, --compare_file=FILE       set the file to be compared
                                with the output
  -s, --safe_python             use py_exe.py as executor
  -p, --pypy3                   use pypy3 instead of cpython,
                                forces safe_python mode
  -n, --no_output               execute without logging
  -h, --help                    displays this
```

Al lado de cada argumento se muestra una pequeña explicación de su funcionamiento. Igualmente, vamos a comentar brevemente cada uno de ellos:

- `-t`: establece, en milisegundos, el tiempo máximo durante el que se ejecutará el programa. Si supera ese tiempo, se terminará su ejecución con el envío de una señal. Si no se indica, el tiempo límite por defecto se establece en 9.999 segundos.

- `-m`: establece, en kibibytes, la memoria máxima que se le permite al programa utilizar. Esta limitación se aplica tanto al *heap* como a la pila del programa. Si se supera este límite, se finaliza la ejecución del programa. Si no se indica, la memoria máxima se establece en 1 GiB.

- `-i` y `-o`: indican, respectivamente, el fichero que contiene la entrada del programa y el fichero donde tiene que escribir su salida. En caso de no estar presente alguno de ellos, se lee / escribe a partir de la entrada / salida estándar.

  Al mismo tiempo que obtenemos ambos ficheros, se realiza el duplicado de la entrada o salida estándar sobre ellos. Con esto conseguimos que el programa que ejecuta simplemente lea y escriba en las estándar, dado que no tiene permisos para escribir en ficheros y tampoco queremos que el usuario se preocupe por dónde leer o escribir.

- `-c`: establece el fichero con el que se comparará la salida del programa. Si no se indica, simplemente no se hace comparación y se acepta su ejecución.

- `-s`: indica al programa que la restricción de llamadas al sistema y el cambio de directorio al *chroot* se realice dentro de un programa auxiliar *Python*. Este programa lo vimos en la sección de Ejecución Segura, donde restringimos las llamadas utilizando directamente *Python*.

- `-p`: indica al programa que la ejecución no se realice con la implementación estándar de *Python*, *CPython*, sino con una alternativa escrita en *Python* puro, *PyPy3*. Dado que esta implementación realiza llamadas al sistema diferentes, esta opción fuerza la opción anterior, ejecutar mediante el intermediario.
- `-n`: la ejecución se realiza sin escribir en consola.
- `-h`: muestra el menú de ayuda.

5.2. Ejecución de *Python*

5.2.1. Compilación

El primer paso que se realiza sobre el programa a ejecutar es compilarlo. *Python* es un lenguaje interpretado, por lo que la fase de compilación no es muy restrictiva y los únicos errores que va a detectar de forma habitual son los de indentación (*Python* depende completamente del indentado para reconocer los bloques de códigos, ya que carece de las típicas llaves de otros lenguajes),

El mismo sistema de ejecución es el encargado de realizar la compilación. En caso de que esta de lugar a algún error, devolverémos un CE y no ejecutaremos el programa. La compilación se realiza mediante la librería de *Python*, *py_compile*(Van Rossum y Drake, 1995, C. 32.10) con el siguiente comando:

```
python -m py_compile file.py 2> compilation_output
```

donde *file.py* es el programa a ejecutar. El resultado de la compilación se guarda en un fichero llamado *compilation_output* el cuál, en caso de que la compilación resultase errónea, nos facilitará el mostrarlo al usuario.

5.2.2. Establecer límites y restricciones

Si la compilación ha resultado exitosa, lo siguiente que tenemos que establecer antes de poder ejecutar son los límites que nos exigen los argumentos. Estos son, el límite de tiempo, el de memoria y las restricciones a llamadas de sistema.

Las limitaciones a la memoria y a las llamadas se realizan tal y cómo hemos visto en los capítulos anteriores, mientras que la del tiempo tenemos que desarrollarla en este apartado. Dado que no es una parte sobre la que vayamos a profundizar, se ha utilizado casi íntegramente esta plantilla[1], adaptándola para ejecutar *Python*.

[1] `https://www.linuxprogrammingblog.com/code-examples/signal-waiting-sigtimedwait`

Dentro del ejecutor, el establecimiento del límite de tiempo es el siguiente:

```
timeout.tv_sec = (time_t)(time_limit_ns / (long)S_TO_NS);
timeout.tv_nsec = time_limit_ns % S_TO_NS;

sigemptyset(&mask);
sigaddset(&mask, SIGCHLD);

if (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0) {
    perror ("sigprocmask");
    exit(1);
}
```

La estructura *timeout* guarda el tiempo límite obtenido de los parámetros en dos campos, segundos y nanosegundos, por lo que tenemos que extraer ambas métricas de nuestro límite en nanosegundos.

A continuación, inicializamos el conjunto de señales (*sigset_t*) *mask* y le añadimos la señal SIGCHLD. Por último, añadimos este nuevo conjunto a las señales bloqueadas con `sigprocmask()`. Esto hará que el proceso ignore esta señal y le permitirá ejecutarse sin problema. Este conjunto de señales bloqueadas son heredadas por los hijos al hacer `fork()` y al cambiar de proceso con un `exec()`, por lo que continuará haciendo su función durante la ejecución de *Python*.

5.2.3. Ejecución

La ejecución comienza obteniendo el tiempo actual, para poder reportar el tiempo total de ejecución. Idealmente, esta tarea se realizaría justo antes de cambiar de proceso a *Python*, pero esto lógicamente va a ocurrir en un hijo, por lo que su asignación no sería visible en el padre.

Para obtener el tiempo actual, utilizamos la función `gettimeofday()`:

```
if(gettimeofday(&starttime,NULL))
    perror("getting time");
```

Lo siguiente es el `fork()`, que nos divide la ejecución en dos partes, la del hijo y la del padre. Este último se queda esperando a que el hijo finalice, o en su defecto a mandarle una señal de SIGKILL si supera el tiempo límite. En otro caso, obtiene el valor de retorno del hijo, que será el del programa *Python* ejecutado.

El hijo por otra parte, se dedicará a preparar el entorno para la ejecución del programa. Si es necesario, copiará el programa a ejecutar sobre el *chroot*. A continuación cambiará directorio raíz por la carpeta creada en la sección de Ejecución segura mediante *chroot* y cargará el filtro de llamadas a sistema sobre sí mismo:

```
char command[512];
sprintf(command, "cp %s root/%s", progpath, progname);
system(command);

chroot("root/");
chdir("/");

if (seccomp_load(ctx) != 0) {
    fprintf(stderr, "ERROR: Couldn't load execution filters.");
    exit(-1);
}
```

En último lugar y según lo que indiquen los parámetros, iniciamos la ejecución del programa *Python* con la configuración adecuada. Tenemos un total de 3 posibilidades:

- Ejecución con *CPython* directa: en donde la tarea de restringir las llamadas y el *chroot* recae sobre el ejecutor, la situación explicada anteriormente.
- Ejecución con *CPython* con un programa intermediario: en lugar de ser el ejecutor el encargado de esta tarea, la relegamos sobre un programa intermedio escrito en *Python*. Este programa lo vimos en la sección de Restricción de llamadas al sistema como más restrictivo, ya que el intérprete está inicializado, lo que nos permite mayores restricciones sobre las llamadas.
- Ejecución con *PyPy3*: en lugar de utilizar el intérprete por defecto de *Python*, *CPython*, el programa se ejecutará con *PyPy3*, la implementación alternativa. Esta opción fuerza el uso del programa intermedio, ya que la inicialización de *PyPy3* utiliza llamadas al sistema que de ninguna manera queremos permitir.

5.3. Resultados de la ejecución

Al finalizar la ejecución del programa, ya sea de forma exitosa o forzada por el sistema, el ejecutor recibe el valor de salida de este. Es responsabilidad del ejecutor recopilar las estadísticas de ejecución del programa finalizado.

En primer lugar, así como antes de la ejecución obtuvimos el tiempo de inicio, obtenemos el tiempo de finalización de la misma forma. Con una simple resta entre ambos (en este caso 2, una para los segundos y otra para los nanosegundos) establecemos el tiempo de ejecución total. En caso de que la finalización ocurriera a causa de superar el tiempo límite, se establece en una variable que se ha alcanzado el límite, al mismo tiempo que terminamos su ejecución con un *kill*.

```
void get_time_ms(){
    int seconds_elapsed = (endtime.tv_sec - starttime.tv_sec);
    long microseconds_elapsed = (endtime.tv_usec - starttime.tv_usec);
    time_elapsed = (long)(seconds_elapsed) * 1000 +
                                    (microseconds_elapsed / 1000);
}
```

En segundo lugar, obtenemos la memoria máxima usada. Este proceso es igual que el explicado en la sección de Medición de memoria, utilizar la función `getrusage` sobre RUSAGE_CHILDREN y el campo `ru_maxrss`. En caso de que la memoria utilizada sea superior a la permitida, marcamos que hemos llegado a un *memory limit*:

```
void get_memory_kb(){
    getrusage(RUSAGE_CHILDREN, &use);
    memory_usage = use.ru_maxrss;
    if(memory_usage * 1024 > memory_limit_bytes)
        memory_limit_reached = 1;
}
```

En tercer lugar, con el valor de retorno y las marcas de límite de tiempo y memoria, podemos establecer si la ejecución fue interrumpida por alcanzar alguno de estos límites, si finalizó correctamente o si finalizó de forma abrupta, pero no a consecuencia de alcanzar ningún límite. Esta tercera causa engloba todo tipo de errores durante la ejecución del propio programa, como por ejemplo dividir por cero.

Este error es conocido como *Run Time Error* o *RTE* y lo vamos a obtener a partir de los campos mencionados:

```
if(returnStatus && !time_limit_reached && !memory_limit_reached){
    if(returnStatus == 159)
        restricted_function = 1;
    run_error_reached = 1;
}
```

Un programa da como resultado *RTE* cuando su valor de retorno no es cero (no sea exitoso) y no haya alcanzado ningún límite.

Con ese mismo valor de retorno, podemos inferir qué tipo de excepción se produjo, con una particular en mente, la de una llamada al sistema prohibida. Concretamente, el retorno del valor *159* nos dice que el programa finalizó a causa de las reglas que impusimos sobre él. Por esto, si el valor es *159*, indicamos que se ha utilizado una función prohibida, en la variable *restricted_function*.

Profundizando un poco en los valores de retorno de un proceso, aquellos superiores a 128 indican qué señal produjo la finalización de su ejecución. A

cada señal en *Linux* le corresponde un número[1], empezando en 1 con la señal SIGHUP. Sabiendo que la señal que finaliza la ejecución cuando se produce una llamada bloqueada es SIGSYS, la número 31, obtenemos 159 como valor de retorno a esperar.

En último lugar, y solo en caso de que la finalización tenga éxito (el valor de retorno sea cero), tenemos que comprobar que la salida del programa es correcta. Esta tarea solo se realiza si un fichero con los datos correctos ha sido proporcionado en los argumentos del ejecutor.

La implementación del comparador es extremadamente sencilla, dada su baja importancia en el proyecto en general. Así, esta se realiza mediante la siguiente función:

```
void check_answer(){
    if(memory_limit_reached || time_limit_reached ||
      run_error_reached || !exists_compare_file)
        return;
    char command[512];
    sprintf(command, "cmp -s %s %s", output_file_str,
                                      compare_file_str);
    int ret = system(command);
    if(ret)
        wrong_answer = 1;
}
```

Comprobamos si la ejecución finalizó correctamente con una disyunción entre los cuatro errores posibles al mismo tiempo que comprobamos la existencia de un archivo a comparar. La comparación se realiza con la herramienta *cmp* de *Linux*, que compara byte a byte los ficheros y muestra por pantalla las diferencias encontradas. Como no queremos saber las diferencias, solo queremos saber si son iguales o diferentes, usamos la opción `-s` para no mostrarlas. Si el valor de retorno es 0, los ficheros son iguales, si es otro valor, estos difieren en al menos 1 byte, por lo que marcamos que la respuesta es errónea.

La herramienta de comparación resulta muy útil en esta situación, donde solo queremos una forma rápida de comprobar si la salida es idéntica al fichero correcto, pero presenta severas limitaciones que no le permitirían funcionar como corrector de un juez decente. La más importante de todas es su completa inflexibilidad en la comparación, lo cual haría comprobar la corrección de problemas que trabajen con números en punto flotante prácticamente imposible.

Una posible mejora sería la de añadir un nuevo parámetro al programa, que le indicase el programa que se va a ocupar de la corrección de la salida del programa. De esta forma, si el problema en particular lo requiere, podemos escribir un comparador específico para él, que se encargue de comprobar la

[1] `https://man7.org/linux/man-pages/man7/signal.7.html`

solución y que devuelva el veredicto como valor de retorno.

5.4. Compilación y Prueba de ejecución

Algo que se comentó brevemente pero no se entró en detalle en la sección de restricción de llamadas al sistema, es el hecho de que no utilizamos la llamada al sistema `seccomp` para realizar las restricciones, sino la librería *lseccomp*. Esto tenemos que indicárselo al compilador, enlazando la librería en la fase de compilación:

```
$ gcc exe.c -o exe -lseccomp -O2
```

Exceptuando este enlazado, el programa no necesita nada más para su compilación y ya está disponible para ejecutar programas. En esta sección vamos a probar el comportamiento básico del ejecutor, los diferentes veredictos que obtenemos y su presentación.

Vamos a empezar las pruebas con un programa vacío:

```
$ sudo ./exe f.py
Settings:
 - Time: 9999000 (ms)
 - Memory: 1048576 (kB)
 - Input file: .
 - Output file: .
 - Compare file: .
 - Safe with python: 0
 - Interpreter: python
 - Executing file: f.py

Veredict: AC
    Time: 29 (ms)
  Memory: 9976 (kB)
```

Otro aspecto importante que tampoco se ha mencionado es la necesidad de ejecutar el programa como superusuario, ya que se necesitan estos permisos para cambiar el directorio raíz con `chroot`.

El ejecutor nos muestra la configuración creada a partir de los argumentos pasados, en este caso con todas las opciones por defecto. Hemos hablado ya de los valores por defecto del tiempo, memoria y ficheros (que se muestran con '.' si no se indican), y nos falta establecer el intérprete de *Python* por defecto, que es *CPython* y si se hace uso del programa intermedio, que por defecto no se utiliza.

Tras la configuración se nos muestra el veredicto, el tiempo y la memoria utilizada por el programa. El veredicto puede tomar los siguientes valores:

- AC: el programa ha finalizado correctamente y, si se ha incluido un fichero para comparar, la salida del programa coincide con la esperada.

Abreviación de *Accepted*.

- WA: el programa se ha ejecutado correctamente, pero la salida producida difiere de la esperada, por lo que se considera incorrecto. Abreviación de *Wrong Answer*.

- TLE: el programa no ha finalizado en el tiempo máximo establecido, por lo que se ha terminado su ejecución antes de tiempo. Abreviación de *Time Limit Exceeded*.

- MLE: el programa ha intentado usar más memoria de la permitida, por lo que ha finalizado su ejecución antes de tiempo. Abreviación de *Memory Limit Exceeded*.

- RTE: durante la ejecución del programa se ha producido alguna excepción, por lo que su ejecución ha finalizado antes de tiempo. Este error puede presentarse por múltiples motivos, como pueden ser dividir por cero, acceder a una posición de un *array* incorrecta, etc. Abreviación de *Run Time Error*.

- CE: se ha producido algún fallo en la compilación del programa, lo que impide su ejecución. Junto a este error se indica en el fichero *compilation_output* el error concreto generado, información que se proporcionará al usuario. Abreviación de *Compilation Error*

- RF: el programa ha utilizado alguna llamada al sistema bloqueada por el entorno. Abreviación de *Restricted Function*.

Los errores de compilación no son muy comunes en *Python* al ser un lenguaje interpretado, y en su gran mayoría resultan ser a causa de mezclar espacios con tabuladores, de indentar incorrectamente bloques o por olvidar los dos puntos después de un *if* o un *while*. Veamos el error que se produce ante alguno de estos casos y qué mensaje nos muestra:

```
if True
    print(1)
```

El programa anterior carece de ':' que abren el bloque del *if*. Si se lo pasamos al ejecutor obtenemos el siguiente veredicto:

```
$ sudo ./exe f.py
Veredict: CE
    Time: 0 (ms)
  Memory: 0 (kB)
```

Ha detectado correctamente que se trata de un error de compilación y guardado la salida en el fichero *compilation_output*:

```
  File "/home/david/Desktop/TEST/f.py", line 1
    if True
           ^
SyntaxError: invalid syntax
```

Python es por lo general un lenguaje benevolente con el usuario y muestra de forma concisa el error que se ha producido. Como ya sabíamos, nos faltan los dos puntos después del *if*.

Capítulo 6

Validación

Con el entorno finalizado, tenemos que asegurarnos de que su funcionamiento es correcto en todo tipo de situaciones. En este capítulo vamos a comprobar que los veredictos obtenidos son los correctos acorde a los parámetros proporcionados y que los límites establecidos se cumplen independientemente de sus valores y del programa ejecutado. También estudiaremos el efecto del entorno sobre el tiempo y memoria y muchos otros aspectos que resultan vitales para la evaluación de los programas.

Algo sobre lo que también vamos profundizar es en el uso de llamadas al sistema. Actualmente, la ejecución de los programas se puede realizar mediante dos ejecutores: el principal, escrito en C y que es menos restrictivo debido a la necesidad de permitir todas las llamadas que hace el intérprete; y un segundo, escrito en *Python* y que actúa como intermediario entre el ejecutor principal y el programa a ejecutar. Este último, al no tener que cargar el intérprete (pues ya lo está), resulta mucho más restrictivo (solo permite 10 llamadas al sistema, mientras que el ejecutor en C permite casi 50).

Es altamente probable que el segundo ejecutor sea demasiado restrictivo durante la ejecución y que programas perfectamente válidos, que no utilicen ninguna función peligrosa, vean su ejecución finalizada al realizar alguna llamada al sistema que hemos bloqueado. Es por eso que en este apartado vamos a prestar especial atención a todos los veredictos *Restricted Function* y obtener la llamada que lo ha provocado.

6.1. Obtención de programas para validar

Para realizar las validaciones necesitamos un conjunto de programas escritos en *Python*, las entradas que esperan estos programas y la salida que se espera que produzcan. Además, es importante contar con un número relativamente elevado de programas y que estos sean variados, preferiblemente de diferentes usuarios con estilos de programación diferente.

Escribir nosotros estos programas no cumpliría con este segundo requisito y aunque lo hiciese, el volumen de programas necesario hace la tarea inabarcable para el tamaño del proyecto. Por estas razones, hemos optado por buscar repositorios de aficionados a la programación competitiva. Una rápida búsqueda pone a nuestra disposición multitud de repositorios llenos de problemas listos para realizar las validaciones[1][2][3][4][5].

Tenemos los programas, ahora nos falta obtener un conjunto de entradas y sus correspondientes salidas para cada uno de ellos. Los programas corresponden a problemas del juez *onlinejudge*, y cada uno de estos problemas cuenta con una entrada y salida de ejemplo para que los usuarios comprueben sus programas antes de realizar un envío. Una posible solución sería intentar extraer de cada problema estas secciones, cosa que finalmente resultaría en un fracaso, pues estas entradas por lo general contienen casos pequeños no indicativos de las situaciones extremas que queremos comprobar.

La solución a nuestros problemas se encuentra en la página *udebug*[6], una página que recopila multitud de ficheros de entrada de gran cantidad de problemas para diferentes jueces en línea, entre los que *onlinejudge* se encuentra. Además, a consecuencia de que las diferentes entradas son proporcionadas por los usuarios, generalmente estas abarcan casos mucho más extremos que los casos de prueba de los propios problemas.

Los puntos positivos de la página no se acaban aquí, ya que la página cuenta también con una *API* la cuál nos permitirá obtener con una simple consulta entradas y salidas para cada uno de los problemas. El uso de la *API* únicamente requiere contactar con el personal de la página para la autorización de uso.

En resumen, se han obtenido un total de 598 programas de 5 repositorios que resuelven 422 problemas, debido a que entre los repositorios hay problemas repetidos. Esto resulta ser un punto positivo, ya que podremos comprobar como el mismo problema puede ser resuelto de formas diferentes, las cuales el entorno tiene que ser capaz de manejar.

6.2. Validación

Para efectuar la validación se ha escrito un *script* que se encarga de ejecutar todos los programas de alguno de los repositorios obtenidos con todas las entradas de las que dispone *udebug* de ese problema. En último lugar, se compara la salida del programa con la esperada, también obtenida

[1] `https://github.com/sajjadt/uvapy`
[2] `https://github.com/secnot/uva-onlinejudge-solutions`
[3] `https://github.com/petabite/UVa`
[4] `https://github.com/jlhung/UVA-Python`
[5] `https://github.com/marceloamancio/acm_uva`
[6] `https://www.udebug.com/`

de *udebug*. El ejecutor nos devolverá un valor de retorno, que se corresponderá al veredicto obtenido del programa.

Veredicto	AC	WA	TLE	MLE	RTE	CE	RF
Código	0	1	2	3	4	5	6

La validación por su parte cuenta con dos conjuntos de programas, uno de entrenamiento utilizado para modificar el entorno hasta que su ejecución sea correcta y un segundo conjunto, de validación, que se utilizará para validar que el entorno funciona correctamente con otro tipo de problemas y que los cambios no solo afectaban al primer conjunto.

La separación en conjuntos es muy sencilla, el repositorio que cuenta con más problemas resueltos se utilizará como conjunto de entrenamiento y el resto de repositorios como validación. El primer conjunto cuenta con 308 problemas, mientras el segundo tiene 290. La existencia de problemas repetidos no afecta demasiado a la validación, ya que al ser realizados por usuarios diferentes, estos tendrán estilos de programación distintos.

6.2.1. Restricción de llamadas

Como ya mencionamos, lo primero que vamos a comprobar es que los diferentes ejecutores permitan todas las llamadas al sistema que se utilicen en programas que no consideremos peligrosos. Por ello, en este primer apartado vamos a prestar especial atención a los veredictos que resulten en *Restricted Function*.

Empezamos validando el entorno por defecto, que restringe las llamadas y hace *chroot* en el mismo programa. Esto resulta en unas restricciones menos severas, ya que necesita permitir todo aquello utilizado en la inicialización del propio intérprete.

```
$ python script TRAIN C
...
TRAIN RESULTS
VER:  AC  WA TLE MLE RTE CE RF
NUM: 803  87   4   0  56   0  0
```

De forma más o menos esperada, el *script* finaliza la ejecución del primer conjunto de problemas sin ningún *RF*, por lo que esperamos que se comporte igualmente en el conjunto de validación.

```
$ python script TEST C
...
TEST RESULTS
VER:  AC   WA TLE MLE RTE CE RF
NUM: 748 102  18   0  70   0  0
```

Como habíamos previsto, el ejecutor por defecto no devuelve ningún *RF* en el conjunto de validación, lo que nos ofrece una confianza bastante elevada de su corrección. A pesar de estos resultados, no debemos confiarnos sobre su validez, pues el conjunto de problemas evaluados palidece ante el volumen de programas que se enfrentaría en una situación real. Es por esto que sería importante estar atentos a este veredicto en un entorno real y evaluar cada situación de forma individual.

A continuación evaluamos el ejecutor que utiliza un programa intermedio en *Python* para restringir las llamadas.

```
$ python script TRAIN S
...
VER:  AC  WA TLE MLE RTE CE  RF
NUM: 723 79   1   0   0   0 147
```

Dado que tenemos un total de **147** veredictos *RF*, en este caso vamos a tener que trabajar un poco más para resolverlos. De nuevo, este comportamiento era más o menos esperado, pues el entorno permite la ejecución de 10 llamadas al sistema, mientras que el ejecutor en C permite casi *50*.

Para obtener la llamada al sistema que ha producido el error, vamos a volver a utilizar una herramienta que ya conocemos, *strace*. Además, hemos modificado el *script* de ejecución de programas para que, en caso de producirse un *RF*, ejecute el siguiente comando:

```
strace -f ./exe problem.py -t 10000 -i input.in -o out -s
    -c output.out 2>&1 | grep -B 1 "killed" | head -1
```

En donde:

- `strace -f`: obtiene la traza de la ejecución, no solo del programa indicado, sino también de sus hijos.
- `./exe problem.py -t 10000 -i input.in -o out -s -c output.out`: invoca la ejecución de *problem.py* con un límite de tiempo de 10 segundos, usando la entrada contenida en el fichero *input.in*, escribir su salida en *out* y compararla con el fichero *output.out*. Además, ejecutar utilizando el programa *Python* de intermediario.
- `2>&1`: redirige la salida de error a la salida estándar.
- `grep -B 1 "killed"`: obtiene la primera línea de todo el texto que contiene la cadena *killed*.
- `head -1`: dada una línea de texto, obtiene la anterior.

En resumen, ejecutamos de nuevo el programa en el entorno con los mismos parámetros, al mismo tiempo que obtenemos su traza. Cuando acaba, obtenemos la línea anterior a la finalización forzosa del programa, que nos

dirá la llamada al sistema que ha provocado esta finalización. En el siguiente fragmento vemos la traza del programa y la llamada que causó el problema:

```
[pid 22736] munmap(0x7f4bd9cbf000, 299008) = ?
[pid 22736] +++ killed by SIGSYS (core dumped) +++
```

Vemos que, efectivamente, se trata de una llamada bloqueada, `munmap`, que si accedemos a su entrada en el manual[1], crea un nuevo mapeo de direcciones virtuales del proceso. Esta llamada se produce cuando *Python* necesita mayor área de memoria, por lo que no es una función peligrosa por sí misma y la debemos permitir. En caso de que la memoria pedida sea muy elevada, el límite de memoria establecido se encargará de ello.

El proceso de obtener las llamadas al sistema realizadas es iterativo, por lo que vamos a iniciar la ejecución de todo el *script* desde el inicio hasta que esta finalice todos los programas correctamente. Una vez hemos llegado a ese punto, hacemos un resumen de las nuevas llamadas permitidas:

- `munmap`: crea un nuevo mapeo de direcciones virtuales.
- `mremap`: expande un mapeo de memoria existente.
- `madvise`: pregunta al *kernel* acerca de un rango de memoria, con el objetivo de mejorar el rendimiento.
- `mprotect`: protege una región de memoria para el proceso.
- `openat`: realiza las mismas acciones que `open` con ligeras diferencias. Al igual que en el programa en C, solo permitimos esta llamada cuando el fichero **no** se abre para escribir (O_WRONLY) o intenta crearlo (O_CREAT).
- `stat`: obtiene información de un fichero. El intérprete de *Python* la utiliza para importar los módulos y por tanto para acceder a la librería estándar. Por esta razón, es completamente necesaria la llamada.
- `newfstatat`: realiza la misma funcionalidad que `stat`, con ligeras diferencias en su uso.
- `getdents64`: lee las entradas de un directorio. Utilizado también en la carga de módulos.

Permitiendo la ejecución de estas llamadas, además de las que ya permitíamos, el primer conjunto de programas se ejecuta completamente sin ningún *RF*.

[1] `https://man7.org/linux/man-pages/man2/mmap.2.html`

```
$ python script TRAIN S
...
TRAIN RESULTS
VER:  AC  WA TLE MLE RTE CE RF
NUM: 803 87   4   0  56   0  0
```

Con estos cambios, el ejecutor de *Python* obtiene exactamente los mismos veredictos que el ejecutor por defecto. Comprobemos si las restricciones actuales permiten la ejecución del conjunto de validación:

```
$ python script TEST S
...
TEST RESULTS
VER:   AC  WA TLE MLE RTE CE RF
NUM: 748 102  18   0  70   0  0
```

Efectivamente, no obtenemos ningún *RF*. Lo último que nos quedaría por validar es el ejecutor de *PyPy3*, que utiliza de forzosamente el intermediario para bloquear las llamadas. Los resultados del conjunto de entrenamiento son los siguientes:

```
$ python script TRAIN P
...
TRAIN RESULTS
VER:  AC  WA TLE MLE RTE CE RF
NUM: 791 86   3   0  70   0  0
```

La razón por la que *PyPy3* obtiene menos *AC*, *WA* y *TLE* que el resto y obtiene más *RTE* es debido a que, al utilizar una versión anterior de *Python*, los módulos `string` y `fractions` no están disponibles, por lo que los programas fallan antes de iniciarse. Obviando este hecho, vemos que no se ha generado ningún veredicto *RF*, por lo que en este apartado damos por bueno su funcionamiento. Para finalizar, comprobemos que la ejecución del conjunto de validación es correcta:

```
$ python script TEST P
...
VER:  AC  WA TLE MLE RTE CE RF
NUM: 741 96  14   0  87   0  0
```

En estos programas volvemos a tener el mismo problema, más *RTE* a consecuencia del uso de módulos que no se encuentran en *PyPy3*. Como lo que realmente nos interesa, el número de *RF*, es cero, pasa la validación.

6.2.2. Efectos del entorno sobre el tiempo y la memoria

Teniendo una mayor confianza en las capacidades del entorno, continuemos con las validaciones. En este apartado comprobaremos si existen diferencias en el tiempo de ejecución y uso de memoria entre el simple uso de

Python, con cualquiera de las implementaciones y la ejecución dentro del entorno, donde a los programas se les establecen límites de tiempo, memoria y llamadas disponibles.

Los programas utilizados para medir las diferencias se corresponden con implementaciones sencillas de algunos algoritmos clásicos, operaciones de lectura / escritura, ordenar una lista y más. En la siguiente tabla se muestran los tiempo obtenidos en la primera parte de las pruebas, en milisegundos. Las tres primeras filas muestran los tiempos de ejecución de la implementación *CPython*, la primera fuera del entorno, la segunda restringiendo las llamadas desde el ejecutor en C (*CPython C* en la tabla) y la tercera utilizando el programa *Python* intermedio para bloquear las llamadas (*CPython S*). Las dos últimas muestran la ejecución del programa utilizando *PyPy3* fuera del entorno en la primera y la ejecución dentro de él (*PyPy3 S* en la tabla) en la segunda:

	I/O		Ordenar	Montículo		Grafos		
	Input	Output		Insertar	Crear	Floyd	Dijkstra	Kruskal
CPython	1.134	1.739	1.611	1.162	650	12.673	3.757	3.464
CPython C	1.116	607	1.899	1.378	910	12.901	4.505	3.720
CPython S	1.101	645	1.968	1.314	990	12.861	4.789	4.043
PyPy3	366	1.274	1.078	966	968	523	2.996	3.979
PyPy3 S	642	732	1.401	1.289	1.468	847	3.187	4.391

Si ignoramos por un momento los resultados de la salida, las diferencias de tiempos de ejecución son más o menos constantes: alrededor de 300 milisegundos en el caso del ejecutor por defecto (en la tabla *CPython C*), unos 450 si añadimos el ejecutor intermediario (*CPython S* en la tabla) y de nuevo unos 300 para *PyPy3*. Estos resultados eran esperados, ya que además de la propia ejecución estamos añadiendo restricciones extra que suponen una sobrecarga.

La prueba de salida, *Output* en la tabla, simplemente escribe un millón de enteros a la salida estándar. La sorprendente diferencia entre la ejecución normal y en el entorno puede ser debida a que, al medir el tiempo en la ejecución normal se ha redirigido la salida a un fichero desde la terminal, mientras que en el entorno se escribe con normalidad a salida estándar, que ha sido redirigida al fichero indicado.

Análoga a la tabla anterior, en la siguiente tenemos la memoria medida en KiB utilizada por cada uno de estos programas durante la misma ejecución:

	I/O		Ordenar	Montículo		Grafos		
	Input	Output		Insertar	Crear	Floyd	Dijkstra	Kruskal
CPython	8.696	8.092	240.056	240.124	240.108	13.568	148.972	94.572
CPython C	9.816	9.696	239.084	239.612	239.152	13.080	148.024	94.060
CPython S	12.652	12.384	243.900	243.716	243.708	17.240	153.012	98.400
PyPy3	68.020	68.284	308.632	328.668	308.124	63.344	133.904	133.592
PyPy3 S	78.364	78.112	319.016	319.000	317.280	74.836	148.856	147.056

La diferencia de memoria utilizada es prácticamente nula en el entorno por defecto, siempre y cuando el programa haga un fuerte uso de la memoria. Así, en las pruebas de entrada y salida donde la memoria utilizada es mínima, existe una diferencia de alrededor de 2.000 KiB. El uso de memoria ejecutando con el intermediario es mucho más regular, utiliza cerca de 4.000 KiB extra en todos los programas. Por último, *PyPy3* es igual de constante, con 10.000 KiB extra de memoria, exceptuando en la inserción en un montículo, que por alguna razón utiliza *menos* memoria en el entorno.

En la segunda parte de las pruebas tenemos la criba de Eratóstenes[1], dos programas que suman raíces cuadradas y programas que emplean vuelta atrás para obtener todos los subconjuntos y permutaciones de una lista. Al igual que antes, primero analizamos el tiempo de ejecución y luego el uso de memoria:

	Números			Vuelta atrás	
	Criba	Sqrt Sum	Sqrt Sum Opt	Subconjuntos	Permutaciones
CPython	1.313	1.669	1.201	1.2327	6200
CPython C	1.348	1.754	1.318	12.720	6.186
CPython S	1.480	1.743	1.387	13.104	6.404
PyPy3	49	209	67	503	881
PyPy3 S	303	447	337	1.448	1.190

Los resultados de esta segunda prueba son muy parecidos a los de la anterior, por lo que no tenemos mucho que comentar sobre ellos. La ejecución en el entorno por defecto es ligeramente más lenta que la del intérprete por sí mismo, usar el intermediario ralentiza un poco más este valor, cosa que también le ocurre a *PyPy3*.

Respecto al uso de memoria, tenemos también resultados muy similares:

	Números			Vuelta atrás	
	Criba	Sqrt Sum	Sqrt Sum Opt	Subconjuntos	Permutaciones
CPython	8.784	403.996	8.332	7.816	8.088
CPython C	10.116	403.088	9.812	9.744	10.048
CPython S	12.472	408.000	12.296	12.668	1.2552
PyPy3	61.264	138.440	57.444	67.716	67060
PyPy3 S	73.964	151.488	73.052	77.616	77.220

Observamos de nuevo los mismos fenómenos. Si el programa utiliza mucho la memoria apenas existe diferencia entre el entorno por defecto y la ejecución estándar de *Python*, mientras que cuando no lo hace, la diferencia se establece en torno a 2.000 KiB. Por otra parte, al igual que en las pruebas anteriores, usar el intermediario penaliza en 4.000 KiB extras y *PyPy3* utiliza alrededor de 10.000 KiB más.

[1] `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

Capítulo 7

Conclusiones

A lo largo de este trabajo hemos conseguido desarrollar de forma satisfactoria un sistema de ejecución de programas en *Python*, de forma que esta se realiza en un entorno seguro ante código malintencionado y que la memoria utilizable esté limitada al valor que le indiquemos.

El entorno final es capaz de ejecutar cualquier programa en *Python*, dándonos la posibilidad de limitar tanto tiempo de ejecución como memoria máxima, así como indicar los ficheros de entrada y salida del programa, y un fichero para realizar la comparación y obtener el veredicto final.

La limitación de memoria finalmente se realizó mediante llamadas al sistema de *Linux*. Esto es suficiente para el alcance objetivo del proyecto, que era el de desarrollarlo para esta misma plataforma, pero sigue siendo una limitación a tener en cuenta.

Respecto a la propia limitación, debemos hablar sobre sus carencias. En primer lugar, tenemos que las limitaciones se realizan sobre el segmento de memoria de datos (RLIMIT_DATA) y sobre el tamaño de la pila (RLIMIT_STACK), mientras que la medición de la memoria máxima obtiene el tamaño máximo de memoria residente en memoria, que es la suma de ambas. En programas que hacen uso intensivo tanto de datos como de la pila, la limitación de memoria no actúa sobre ellos hasta que algún tamaño de memoria supere el establecido, lo que teóricamente permitiría a los programas llegar a utilizar el doble de memoria del permitido, la mitad en los datos y la otra en la pila. Con un ejemplo:

- Si establecemos el límite en 100 MiB, esta limitación se aplica sobre la memoria de datos en 100 MiB y sobre la memoria de pila en 100 MiB. El sistema operativo no finalizaría la ejecución del programa hasta que alguna de estas limitaciones se alcance, permitiendo que un programa que utilice 90 MiB de datos y 90 MiB de memoria, a pesar de ser superior a los límites establecidos por separado.

Esta carencia solo afecta a la propia ejecución, ya que el programa ejecu-

tor obtiene por separado la máxima usada y si es superior al límite, devuelve un *Memory Limit Exceeded*.

La restricción de llamadas al sistema utilizando `seccomp` depende fuertemente de que su ejecución se realice en un *Linux* que comparta las mismas llamadas que el utilizado para desarrollar el proyecto. Para las restricciones también es importante la versión de Python con la que estamos ejecutando, pues un ligero cambio sobre cómo realiza ciertas funciones podría hacer que el entorno deje de ser capaz de ejecutarlo. Esto podría ocurrir si se cambia la utilización de una llamada al sistema por otra que realiza virtualmente la misma funcionalidad, como por ejemplo el uso de `openat` y `open`.

Respecto a este último punto, comentar que, en el sistema de ejecución final, tal y como vimos en la fase de validación, se permite la apertura de ficheros siempre y cuando esto sea únicamente para lectura. Con esto, un usuario sería capaz de abrir ficheros en el entorno y leer sus contenidos, lo cual consideremos que es inofensivo por dos motivos:

- En primer lugar, abrir los ficheros es lo único que sería capaz de hacer, no se permite el crearlos ni escribir, ni ejecutar nada dentro del entorno.

- En segundo lugar, dado que se ejecuta sobre un directorio raíz que lo único que contiene es lo necesario para ejecutar *Python*, no existen ficheros en el mismo que resulten en ninguna vulnerabilidad del entorno.

Vemos como esta debilidad del entorno es solucionada en parte por *chroot*. El entorno de *chroot* desarrollado permite la ejecución completa del intérprete de *Python* en su interior y solamente del mismo intérprete, pues no contiene ninguna otra herramienta que encontraríamos habitualmente en *Linux*.

A su vez, esto nos ha permitido limitar los módulos que el usuario tiene a su disposición. Esto lo conseguimos durante la creación del *chroot*, obteniendo las dependencias y ficheros que abre un programa en *Python* encargado de importar todos los módulos que vamos a permitir. Estos fueron seleccionados manualmente desde la colección de la librería estándar de *Python*(Van Rossum y Drake, 1995), en base a su utilidad y necesidad en la programación competitiva. Así, los módulos que ofrecen las diferentes estructuras de datos están disponibles, mientras que aquellos referentes a la ejecución paralela y el multihilo no lo están. La lista completa de módulos permitidos se encuentra en el fichero *modules.py*.

Finalmente, en la fase de validación obtuvimos un conjunto moderadamente grande de problemas con entradas y salidas para los mismos y probamos a ejecutarlos dentro del entorno. Esto nos proporcionó información vital para el proyecto:

- En primer lugar, nos obligó a revisar las llamadas al sistema restringidas, ya que algunos programas que consideramos eran adecuados resultaban finalizados por realizar llamadas que no permitíamos. Particularmente, terminamos permitiendo la ejecución de varias llamadas relacionadas con el espacio de memoria, para aumentarlo o disminuirlo. Esto ocurre en problemas que, por ejemplo, obtienen un *array* de la entrada, y su tamaño varía según el caso de prueba. Esto provocaba que en cada caso de prueba la memoria necesitada fuera diferente, resultando en las llamadas que estaban bloqueadas.

- En segundo lugar, nos ofreció información acerca de las penalizaciones resultantes de ejecutar los programas en el entorno, respecto a ejecuciones normales por los propios intérpretes. Los resultados han sido satisfactorios, ya que el entorno solo añade una pequeña constante temporal y de memoria para cada tipo de ejecución, un efecto aceptable considerando las utilidades y seguridad que nos ofrece.

7.1. Trabajo futuro

Una de las metas que finalmente no pudo cumplirse debido a la falta de tiempo era la de implementar este sistema de ejecución sobre el juez *¡Acepta el reto*, incorporando *Python* a la lista de lenguajes disponibles.

Lo primero que necesitaríamos es continuar realizando pruebas de validación sobre el entorno, para asegurarnos de que las llamadas permitidas finalmente son necesarias y suficientes para la ejecución de cualquier programa en *Python* que queremos permitir.

En segundo lugar tendríamos que abordar la tarea de establecer la memoria límite de cada problema. En este momento ninguno de los problemas del juez es resoluble en *Python* con los límites actuales. Los problemas que no hacen uso excesivo de la memoria tienen un límite de 4096 KiB, mientras que *Python* utiliza alrededor del doble solo para iniciarse. Una posible solución consistiría en únicamente medir la memoria utilizada por encima del consumo base del intérprete, cosa ya implementada en el juez para medir la memoria usada por *Java*.

Esto no solucionaría el problema de la memoria, pues en los problemas que sí hacen uso intenso de la memoria (por ejemplo problemas donde necesitamos guardar 1 millón de enteros), *¡Acepta el reto* asume que cada uno de ellos va a ocupar 4 bytes, por lo que el `array` de 1 millón ocuparía 4 MiB. Sin embargo y cómo hemos visto en el Estado del arte, en *Python* cada entero ocupa 28 bytes, el *array* ocuparía 26.7 MiB y tendríamos un MLE. Una posible solución para estos problemas es la de asignar al límite de memoria de *Python* superior, acorde a este hecho, para igualar sus capacidades con el resto de lenguajes. Esto podría ser implementado encontrando un multi-

plicador sobre el límite original que permita que las soluciones eficientes en memoria no lo superen y que las no eficientes obtengan MLE.

Conclusions

Throughout this work we have successfully developed an execution system of *Python* programs, capable of running them safely against malicious code and restricting the maximum usable memory to any given value.

The final environment is able to run any *Python* program, providing us with the possibility of limiting both execution time and maximum memory, as well as making the program read from a given file and write into another, and a third file to perform a comparison between it and the program output, to obtain the final verdict.

The memory limitation was finally done using *Linux* system calls. This is enough as it is the scope of the project, but it is still a limitation to bear in mind.

Regarding the limitation itself, we need to talk about its shortcomings. First of all, the memory limitations affect separately the segment of data memory (RLIMIT_DATA) and the stack of the program (RLIMIT_STACK), while the memory measurement obtains the maximum resident set size, which is approximately the sum of both. In programs that make intensive use of both data and the stack, the memory-limitation system does not interrupt their execution until one of them surpasses the established limit, which would theoretically allow the use twice as much memory as allowed, half on the data and half on the stack. As an example:

- Given a limit of 100 MiB, the environment limits the data memory to 100 MiB and the stack memory to also 100 MiB. The operating system will not interrupt the program execution until one the the sections grows to more than the limit, allowing a program to use 90 MiB of data and 90 MiB of stack, which is way higher than the limit we imposed.

This shortcoming only affects the execution itself, since the execution program obtains separately the peak memory and returns the verdict *Memory Limit Exceeded* when it is higher than the limit.

System calls restriction using the `seccomp` library is also strongly dependent on a *Linux* platform that shares the same calls as the one used throughout the project. For the restrictions, the version of *Python* is also very important, since a slight change in the execution of certain functions

could make the environment no longer capable of running them. This could happen if the use of a system call is changed to another that performs virtually the same functionality, such as the use of `openat` or `open`.

Regarding this last point, the final execution system, as we saw in the validation phase, allows programs to open files, as long as it is for reading only. With this, a user would be able to freely open files and read their contents, which we consider to be harmless for two reasons:

- Firstly, open and reading the files is the only thing allowed, neither writing nor creating new files is allowed, much less executing anything inside the environment.
- Secondly, given that the execution takes place inside a directory root that only contains the bare minimum to execute the *Python* interpreter, there are no files inside it worthy of protection.

This weakness of the environment is solved in part by the *chroot*. The *chroot* environment in which the programs end running allows the complete execution of the *Python* interpreter, and only of the interpreter itself, as it does not contain any other tool that we would usually find in *Linux*.

At the same time, *chroot* has allowed us to restrict the modules that users hace at their disposal. This is achieved during the creation of the *chroot* directory, obtaining the dependencies and files opened by a *Python* program in charge of importing every allowed module. These were manually selected from the *Python* standard library, based on their utility and need in competitive programming. Modules chosen are for example all the data structures, while those used for parallel execution and multithread are not available. The complete list of allowed modules can be found in the file *modules.py*.

Finally, in the validation phase we obtained a moderately large set of problems with inputs and outputs for them and tested running them inside the environment. This provided us with vital information for the project:

- First, it forced us to review the restricted system calls, as some programs of the training sets which we considered adequate were being killed for using syscalls that we did not allow. In particular, several system calls related to enlarging and shrinking the memory space owned by the program. This happened in problems which, for example, read and store an *array* from the input, and its size varies in each input. This caused that in each test case the memory needed was different, resulting in the execution of systems calls that are now allowed.
- Second, it offered us information about the penalties of executing the programs inside the environment, in contrast with a normal execution

using only the *Python* interpreter. The final results have been satisfactory since the environment adds just a small temporal and memory constant for each kind of execution, an acceptable effect considering the utilities and security that it offers.

Future work

One of the goals that finally could not happen due to lack of time was to implement this execution system into the *¡Acepta el reto!* judge, adding *Python* to the list of available languages.

The first thing we would need is to continue performing validation test on the environment, to make sure that the allowed system calls are necessary and sufficient for the execution of any *Python* program that we want to allow.

Second, we would have to tackle the task of establishing the memory limit for each problem. At this moment none of the problems on the judge are solvable in *Python* with their current limits. Problems that do not use a lot of memory are capped at 4096 KiB, while *Python* uses about twice as much just for startup. A possible solution for those would be to only measure the memory used above the base consumption of the interpreter, something already implemented in the judge to measure the memory used by *Java*.

This would not solve the overall problem, because in problems that do make intensive memory use (such as reading and storing 1 million integers) *¡Acepta el reto!* assumes that each of them takes 4 bytes, so the whole `array` would be 4 MiB. However and as we have seen in the second chapter, each integer in *Python* uses 28 bytes, so the million integers would take 26.7 MiB, obtaining a MLE. To solve this we could establish a higher memory limit for *Python* to match its capabilities with the other languages. This could be implemented by finding a multiplier on the original limit that allows memory efficient solutions to pass and non efficient to obtain MLE.

Bibliografía

Bolz, C. F., Cuni, A., Fijalkowski, M. y Rigo, A. Tracing the meta-level: PyPy's tracing JIT compiler. En *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, páginas 18–25. 2009.

Corbet, J. Seccomp and sandboxing. *LWN. net, May*, vol. 25, 2009.

Elo, A. E. *The rating of chessplayers, past and present*. Arco Pub., 1978.

Foord, M. J. y Muirhead, C. *IronPython in action*. Manning, 2009.

Friedl, S. Go directly to jail: Secure untrusted applications with chroot. *Linux Magazine*, páginas 2002–12, 2002.

Gómez-Martín, P. P. y Gómez-Martín, M. A. ¡ Acepta el reto!: juez online para docencia en español. *Actas de las Jornadas sobre Enseñanza Universitaria de la Informática*, vol. 2, páginas 289–296, 2017.

Juneau, J., Baker, J., Wierzbicki, F., Muoz, L. S., Ng, V., Ng, A. y Baker, D. L. *The definitive guide to Jython: Python for the Java platform*. Apress, 2010.

Kamp, P.-H. y Watson, R. N. Jails: Confining the omnipotent root. En *Proceedings of the 2nd International SANE Conference*, vol. 43, página 116. 2000.

Looi, W. Analysis of Code Submissions in Competitive Programming Contests. 2018.

Lutz, M. *Learning Python: Powerful object-oriented programming*. O'Reilly Media, Inc., 2013.

Mitchell, M., Oldham, J. y Samuel, A. *Advanced Linux programming*. New Riders Publishing, 2001.

Nilsson, S. *Heapy: A memory profiler and debugger for Python*. Proyecto Fin de Carrera, 2006.

REVILLA, M. A., MANZOOR, S. y LIU, R. Competitive learning in informatics: The UVa online judge experience. *Olympiads in Informatics*, vol. 2(10), páginas 131–148, 2008.

ROGHULT, A. Benchmarking Python Interpreters : Measuring Performance of CPython, Cython, Jython and Pypy. 2016.

VAN ROSSUM, G. Python (programming language) 1 CPython 13 Python Software Foundation 15. 2010.

VAN ROSSUM, G. y DRAKE, F. L. Python library reference. 1995.

VAN ROSSUM, G., DRAKE, F. L. ET AL. Python reference manual. 2000.